Towards Correct Programs in Practice

Proving Functional and Non-Functional
Properties by means of Program Analysis

Alejandro N. Tamalet

Radboud University Nijmegen

ISBN/EAN: 978-94-6191-086-8
IPA dissertation series: 2011-20

Typeset using LaTeX 2_ε
Cover design: Naimé Rubino
Translation of the Dutch summary: Ken Madlener
Printed by: Ipskamp Drukkers

Nederlandse Organisatie voor Wetenschappelijk Onderzoek

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Part of this research was supported by the Netherlands Organization for Scientific Research (NWO) under project number 612.063.511.

Towards Correct Programs in Practice

Proving Functional and Non-Functional Properties by means of Program Analysis

Een wetenschappelijke proeve op het gebied van de Natuurwetenschappen,
Wiskunde en Informatica.

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus, prof. mr. S.C.J.J. Kortmann,
volgens besluit van het college van decanen
in het openbaar te verdedigen op maandag 28 november 2011
om 13:30 uur precies

door

Alejandro Néstor Tamalet

geboren op 28 juni 1978,
te Rosario, Argentinië.

Promotor:

Prof. dr. Marko C.J.D. van Eekelen

Copromotor:

Dr. Olha Shkaravska

Manuscriptcommissie:

Prof. dr. Bart Jacobs	
Prof. dr. Ricardo Peña	Complutense University of Madrid.
Dr. Marieke Huisman	University of Twente.

Towards Correct Programs in Practice

Proving Functional and Non-Functional Properties by means of Program Analysis

An academic essay in Science.

Doctoral thesis

to obtain the degree of doctor
from Radboud University Nijmegen
on the authority of the Rector Magnificus, Prof. dr. S.C.J.J. Kortmann,
according to the decision of the Council of Deans
to be defended in public on Monday, 28 November, 2011
at 13:30 hours

by

Alejandro Néstor Tamalet

born in Rosario, Argentina
on 28 June 1978.

Supervisor:

Prof. dr. Marko C.J.D. van Eekelen

Co-supervisor:

Dr. Olha Shkaravska

Doctoral Thesis Committee:

Prof. dr. Bart Jacobs	
Prof. dr. Ricardo Peña	Complutense University of Madrid.
Dr. Marieke Huisman	University of Twente.

Summary

In a modern society we interact with software on a daily basis. We use software not only when we use our personal computers or mobile phones, but also when we drive our cars or watch television. It is not far-fetched to say that our lives rely on the correct functioning of critical software operating in satellites, aircraft, medical equipment and power stations, among others. To ensure the correctness of such software, which is often extremely complex, a rigorous framework is needed in which properties can be expressed, analysed, and ultimately verified. Program analysis combined with formal methods provides a set of mathematical techniques that allow us to rigorously specify and verify properties.

This thesis presents a number of program analysis techniques and frameworks that aid in proving programs correct. It covers both functional properties, i.e., related to the concrete behaviour of a software system, as well as non-functional properties, i.e., concerning the overall system.

We start with an introduction where we put our research in context and lay down the basis for the following chapters, which are divided in two parts.

The first part concerns modelling properties of programs and proving them correct. All the results in this part have been formally established using the PVS theorem prover.

In Chapter 2 we analyse the inter-process communication (IPC) subsystem of the Fiasco microkernel. One of the difficulties in reasoning about concurrent software is dealing with the huge number of possible states that the system can be in. To this end we develop the *preemption abstraction*, which is a technique for the verification of one component of a concurrent system that yields a sequential abstract system. This sequential system is amenable to be analysed in a theorem prover. In a case study, we use the preemption abstraction to prove some properties and also to uncover a bug that could crash the studied system.

Chapter 3 describes a technique to verify that a program satisfies a security policy described by a security automaton. A security automaton is typically used to monitor a program and stop it as soon as the property it represents is breached. As an example, we work with a property that limits the number of SMSs that a mobile application can send. We describe a procedure that generates JML annotations that implement a monitor of the security property into the application and we prove that if monitoring does not reveal a security violation, the generated annotations are respected by the program. The correctness proof

reveals several subtleties that one must consider with respect to the translation process and the program requirements.

In Chapter 4 we develop a framework to reason about how an assignment can affect a recursive data structure. We first develop a number of rules that describe when and how a path in the heap can be modified (or not) by an assignment. We then use these rules to prove properties about other data structures. A key aspect of this approach is that by applying these rules we do not need to reason inductively: the induction is encapsulated in the rules.

Part II is concerned with resource analysis, in particular we try to statically determine the size of the output of a function definition as a polynomial of the size of its arguments. In Chapter 5 we provide an extensive look at the works in this field.

Chapters 6 introduces a size-aware type system for a first-order functional language with algebraic data types, where types are annotated with polynomials over size variables. We define how to generate typing rules for each data type, provided its user defined size function meets certain requirements. As an example, a program for balancing binary trees is type checked. The type system is shown to be sound with respect to the operational semantics in the class of shapely functions. Type checking is shown to be undecidable, however, decidability for a large subset of programs is guaranteed.

In Chapter 7 we overcome the main restriction of the type system of the previous chapter, i.e., the requirement of shapeliness, by means of a collected size semantics for strict functional programs over lists. The collected size semantics of a function definition is a multivalued size function that collects the dependencies between every possible output size and the corresponding input sizes. Such functions are defined by conditional rewriting rules generated during type inference and they are used as type annotations. Using collected size semantics we are able to infer non-monotonic lower and upper polynomial bounds for many functional programs. As a feasibility study, we use the inference procedure to infer lower and upper polynomial size-bounds on typical functions of a list library.

The thesis ends with a Conclusions chapter where we revise the results obtained and discuss about possible ways to integrate them.

Samenvatting

Interactie met software is bijna niet meer weg te denken in de hedendaagse samenleving. We maken niet alleen gebruik van software als we onze computers of mobiele telefoons gebruiken, maar ook als we in een auto rijden of televisie kijken. Het is daarom niet vergezocht te zeggen dat onze levens afhangen van het correct functioneren van software in satellieten, vliegtuigen, medische gereedschappen, kerncentrales, en dergelijke. Om zeker te zijn van de correctheid van zulke software, die vaak zeer complex is, is een rigoureus raamwerk nodig waarin eigenschappen kunnen worden uitgedrukt, geanalyseerd, en uiteindelijk worden bewezen. Analyse van programma's gecombineerd met formele methoden geeft een verzameling wiskundige technieken die ons in staat stellen op een rigoureuze manier eigenschappen te specificeren en te controleren.

Dit proefschrift presenteert een aantal technieken en raamwerken die ondersteuning bieden bij het correct bewijzen van deze programma's. Het proefschrift beschouwt functionele eigenschappen, dat wil zeggen, gerelateerd aan het concrete gedrag van een software systeem, en daarnaast ook niet-functionele eigenschappen, dat wil zeggen, eigenschappen die te maken hebben met het globale gedrag van het systeem.

Het proefschrift begint met een introductie waarin we het onderzoek in context plaatsen en de basis leggen voor de daarna volgende hoofdstukken, waarbij gebruik is gemaakt van een tweedeling onder de hoofdstukken.

Het eerste deel betreft het modelleren van eigenschappen van programma's en het correct bewijzen van deze eigenschappen. Alle resultaten van dit deel zijn geformaliseerd in de bewijsassistent PVS.

In hoofdstuk 2 wordt het "inter-process communication (IPC)" deelsysteem van de Fiasco microkernel geanalyseerd. Eén van de moeilijkheden in het redeneren over parallelle software is de grote hoeveelheid toestanden waarin het systeem zich kan bevinden. Om deze reden wordt er een techniek genaamd "preemption abstraction" geïntroduceerd. Dit is een techniek om één component van een parallel systeem te verifiëren door middel van het creëren van een sequentieel abstract systeem. Dit sequentiële systeem kan in een bewijsassistent worden geverifieerd. In een case study wordt gebruik gemaakt van preemption abstraction om enkele eigenschappen te bewijzen en mede een bug te vinden dat mogelijk het systeem zou kunnen laten crashen.

Hoofdstuk 3 beschouwt een techniek om te verifiëren dat een programma voldoet aan veiligheidseigenschappen die beschreven worden door security automaten. Een security automaat wordt typisch gebruikt om een programma in de gaten te houden en het programma te stoppen zodra de bijbehorende eigenschap wordt verbroken. Een voorbeeld hiervan is een applicatie op een mobiele telefoon die het maximale aantal SMS berichten dat wordt verzonden beperkt. Hoofdstuk 3 beschrijft een procedure die JML annotaties genereert die een monitor van de eigenschap implementeren. Daarnaast wordt bewezen dat als de monitor de veiligheidseigenschap niet onderbreekt, dan worden de gegenereerde annotaties gerespecteerd door het programma. Het correctheidsbewijs illustreert verschillende subtiliteiten waar men rekening mee moet houden bij het vertalingsproces en het formuleren van programma eigenschappen.

In hoofdstuk 4 wordt een raamwerk ontwikkeld waarmee geredeneerd kan worden over hoe een toewijzing van een waarde aan een variabele een recursieve datastructuur kan beïnvloeden. Eerst worden een aantal regels ontwikkeld die beschrijven hoe en wanneer een pad in een heap kan veranderen (of juist niet wordt veranderd) door een toewijzing. Daarna wordt van deze regels gebruik gemaakt om eigenschappen over andere datastructuren te bewijzen. Een belangrijk aspect van de benadering is dat er niet inductief over de datastructuren hoeft te worden geredeneerd: de inductie is verborgen in deze regels zelf.

Deel 2 van het proefschrift gaat over het analyseren van het verbruik van "resources", waarbij de nadruk wordt gelegd op het statisch analyseren van de grootte van het resultaat van een functie, gedefinieerd als een polynoom in de grootte van de argumenten. In hoofdstuk 5 wordt een uitgebreid overzicht gegeven van het werk in dit vakgebied.

Hoofdstuk 6 introduceert een type systeem dat bewust is van maten voor een eerste-orde functionele taal met algebraïsche datatypes, waarbij de types geannoteerd zijn met polynomen over variabelen die de maten uitdrukken. We definiëren hoe type afleidingsregels kunnen worden gegenereerd voor elk datatype, gegeven dat de functie die de maat uitdrukt, gedefinieerd door de gebruiker, aan bepaalde eigenschappen voldoet. Als een voorbeeld hiervan wordt een programma dat binaire bomen balanceert gecontroleerd door middel van een typesysteem. Er wordt aangetoond dat dit typesysteem sound is met betrekking tot de operationele semantiek in de klasse van shapely functies. Aangetoond wordt dat het controleren van types in het algemeen onbeslisbaar is. Wel kan voor een belangrijk deel van alle programma's worden aangetoond dat het wel beslisbaar is.

In hoofdstuk 7 verbeteren we het belangrijkste gebrek aan het typesysteem van het voorgaande hoofdstuk, dat wil zeggen, de vereiste dat de functies shapely zijn, door middel van een "collected size semantics" voor stricte functionele programma's over lijsten. De collected size semantics van een functiedefinitie is een meerwaardige functie die de afhankelijkheden tussen elke mogelijke maat van uitvoer en de maat van de bijbehorende invoer bijhoudt. Zulke functies worden gedefinieerd door conditionele herschrijf regels die gegenereerd worden gedurende de inferentie van types, en worden gebruikt als type annotaties. Door middel van collected size semantics zijn we in staat polynomiale niet-monotone onder- en

bovengrenzen te bepalen voor een grote klasse functionele programma's. Om de toepasbaarheid te controleren, wordt de inferentieprocedure getest om de polynomiale onder- en bovengrenzen af te leiden van de functies die men tegenkomt in een typische bibliotheek voor lijsten.

Het proefschrift eindigt met een hoofdstuk waarin conclusies worden getrokken en de behaalde resultaten worden herbeschouwd. Ook wordt bediscussieerd hoe de resultaten kunnen worden geïntegreerd.

Acknowledgements

I cannot allow this thesis to be published without thanking all the people that helped me in many ways during these years.

First of all, I would like to thank my supervisor, Marko van Eekelen, for all his support, his excellent guidance and for being there whenever I need him. Next, I want to thank my copromotor Olha Shkaravska for all the insightful talks we had in the blue sofas, for staying working with me until very late the nights before a paper submission and for always being so cheerful. I also want to thank Sjaak Smetsers who cosupervised me during the first year, and Bart Jacobs and Erik Poll who were in charge of the department and made sure everything was running smoothly.

I wish to thank heartily the members of the reading committee Bart Jacobs, Marieke Huisman and Ricardo Peña. This work has greatly benefited from their insightful remarks and suggestions. I am also deeply grateful to the coauthors of my publications: Erik Schierboom, Hendrik Tews, Ken Madlener, Marieke Huisman, Marko van Eekelen, Sjaak Smetsers and Olha Shkaravska. I have learnt a lot from you.

Special thanks to Marieke Huisman who, while supervising my internship in France, recommended me for this position and introduced me to PVS among other things.

Before I thank the people that made my stay in Nijmegen very enjoyable, let me tell a short story that in a way describes the *gezellig* atmosphere of the city and the Digital Security group. After giving a talk in my candidate interview, everything was arranged for me to start the PhD in a couple of months. That same day a member of the group, who had never seen me before, invited me to his wedding party. Since I had already achieved my goal, that night I started celebrating and I remembered Marieke's words: *they will try to get you drunk, and you should let them!* Lets just say that I passed the test... with flying colours.

I would like to thank the secretaries of the group Maria and Desiree, who many times went outside their duties to help me get my work done. I also want to thank Rony and Engelbert for their continuous technical support and for carrying out the essential task of filling the coffee machine.

People at the Digital Security department have always been kind and cheerful to me. They make it hard to think of the place as work. Many thanks to Ana,

Benjamin, Chris, Christian, Flavio, Gerhard, Ichiro, Jaap-Henk, Jasper, Jorik, Julien, Lejla Leonard, Lukasz, Miguel, Pin, Rody, Roel and Wojciech.

I want to express my gratitude to Anne Marie van Lanen for encouraging me to be a maths tutor for students at Arnhem International School. Many thanks to all my students, in particular to Marki, Klaud and the von Ketelhodt family. I also would like to thank Maud and Alex for renting me a typical Dutch attic room and for always being nice to me.

I consider myself really lucky to have met so many wonderful people during these years. I cannot continue without thanking Hilje spending *four* Christmas nights with me and to all the people with whom I have shared unforgettable moments. At risk of forgetting mentioning someone, I would like to thank Alex, Amaru, Annika, Christian, Cristina, Daniela, Daphne, Davide, Deniz, Elena, Elke, Fabio, Francesco, Helene, Jisk, Jordan, Lolle, Marieke, Nadine, Nina, Rostand, Shankar and Vicenç.

It would be unfair to forget the people that helped me from the other side of the world. My next paragraphs are to them.

Jamás hubiera llegado donde llegué sin el apoyo incondicional de mis padres. No es fácil que un hijo se ausente durante tanto tiempo, pero ellos siempre han querido lo mejor para mí. Les agradezco con todo mi corazón a mi viejo Hugo y mi madraza Erica.

Las cosas se hacen mucho más sencillas cuando uno tiene una familia y amigos que lo apoyan y le dan fuerzas. Son realmente demasiados para nombrarlos a todos, pero puedo dejar de agradecerle a mis hermanos Iván, Cristian y Néstor, a mis suegros Norma y Victorio, a mis cuñados Daniel, Ana Clara y Anahí y a sus familias (aunque ya son mías también), a mis preciosos sobrinos Santiago y Julieta, y a mi tía y madrina Gisela.

Un párrafo aparte merece mi hermano Sebastián por ser un modelo a seguir en mi vida. Gracias Oreja por ser como sos.

But there is one person I would like to thank in particular and to whom I dedicate this work. To my wife and partner in life, Liliana.

Al amor de vida, Liliana.

Alejandro N. Tamalet
Rosario, October 2011.

Table of Contents

Chapter 1

Introduction

Nowadays software plays a critical role in almost every facet of our daily life – from watching television, to driving our cars, to working in our offices. Some of these systems are *safety-critical*. Think of the anti-lock brake system (ABS) in your car or the control systems used in avionics and trains. Failure of this kind of software could have catastrophic consequences for human life. Another kind of critical software is the one used in embedded systems, like mobile phones. Software errors in these pervasively distributed systems may not threaten life, but can cause losses of several millions dollars. Hence, being able to guarantee that software programs meet some required properties is of paramount importance.

This thesis covers several topics that belong to the broad area of proving both *functional* and *non-functional* program properties by means of *program analysis*.

Functional properties (or requirements) are related to the concrete behaviour of a software system: they describe the possible outputs of a concrete component for some given input data. Some examples in this thesis are the security automaton of Chapter 3 that restricts the number of SMS messages that can be sent by a mobile phone and the annotations to a list reversal algorithm in Chapter 4 that are used to prove (without using induction) that it indeed reverses lists. On the other hand, non-functional properties concern *overall* characteristics of the system rather than specific behaviours. Examples in this thesis are the absence of deadlock proved for the inter-process communication (IPC) subsystem of a microkernel in Chapter 2 and the typing judgements that describe a size relation between the input and the output of functions in Chapters 6 and 7.

Program analysis is concerned with the study of (mostly) automatic techniques for obtaining predictive information about the dynamic behaviour of programs. The analysis should obtain sound information with respect to the program semantics, that is, obtain approximations that hold for all executions. This means that any approximation must be conservative, i.e., err on the safe side.

The initial motivation for program analysis is to gather information to enable compiler optimisations, for instance, avoiding redundant computations, e.g., by

reusing available results or by moving loop invariant computations out of loops, or avoiding superfluous computations, e.g., of results known to be not needed or results already known at compile-time. This kind of techniques is the driving force behind the recent advances in JavaScript engines, which are at the heart of modern Internet browsers. But program analysis has a wide range of applications, for example, verifying that software respects both safety properties ("something bad will not happen") and liveness properties ("something good will eventually happen") [OL82], aiding to detect errors, validating software received from sub-contractors, allowing execution of foreign code in an untrusted environment and aiding in transformations of data formats (e.g., solving the Year 2000 problem).

The most obvious benefit of program analysis, is that it allows early detection of some programming errors. Errors that are detected early can be fixed immediately, rather than lurking in the code to be discovered much later, even after the program has been deployed or the original programmer is no longer available. Moreover, errors can often be pinpointed more accurately during a static analysis than at run-time, when their effects may not become visible until some time after things begin to go wrong. For a comprehensive introduction to the area of program analysis, we refer the reader to the textbook of Nielson, Nielson and Hankin [NNH99].

One of the guiding principles for our research has been practical applicability. This is a goal shared with the Laboratory for Quality Software (LaQuSo), who sponsored some of this research. This can be seen, for instance, in Chapter 2, where we analyse the inter-process communication subsystem of a well-known microkernel, in Chapter 3 where we advocate the use of state machines to easily describe properties, which are later automatically inlined into the program and enforced at run-time, or in the Java implementation of the type checker of Chapter 6. We hope to have made a contribution towards proving programs correct in practice.

1 Contributions and Organisation of this Thesis

This section describes the organisation of the remaining of this thesis and highlights the contributions of the author.

This work is divided in two parts that reflect the projects the author has been part of, namely the *Laboratory for Quality Software* (LaQuSo)[1]. and the *Amortised Heap Analysis* (AHA)[2]. Except Chapter 5, which describes relevant work on resource analysis, each of the following chapters is based on a publication in an international conference. They are extended with explanations and examples that were not included in the original publications due to space limitations. These chapters are self-contained, i.e., with its own introduction, related work

[1] LaQuSo is a joint activity between Eindhoven University of Technology and Radboud University Nijmegen. See `http://www.laquso.com/`

[2] See `http://resourceanalysis.cs.ru.nl/`

and conclusions so that they can be read any order. It is, however, recommended to read Chapter 6 before reading Chapter 7.

Part I has three chapters that have as common topic the use of a proof assistant, concretely PVS, to model and prove properties of programs.

Chapter 2 presents the *preemption abstraction*, an abstraction technique for lightweight verification of a sequential component of that is part of a concurrent system where different components of that system are permitted to interfere with each other. The preemption abstraction yields a sequential abstract system that can easily be described in the higher-order logic of a theorem prover. One can therefore avoid the cumbersome and costly reasoning about all possible interleavings of state changes of each system component. This technique is used to model the IPC subsystem of the Fiasco microkernel. Two relevant properties of the model were proved. On the attempt to prove a third property, namely that the assertions in the code are always valid, a bug that could potentially crash the whole system was discovered.

As part of his master thesis, Erik Schierboom, first author of this work, made the first version of the PVS formalisation and uncovered the bug. My main contribution to this work was a generalisation of the concrete case study to the broader notion of preemption abstraction and improving the PVS specification. I presented this work at *14th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2009)* on November 2 2009 in Eindhoven, The Netherlands. This chapter is based on the publication [STT+09].

Chapter 3. Security automata are a convenient way to describe security policies. Their typical use is to monitor the execution of an application and to interrupt it as soon as the security policy is violated. In this chapter we aim at developing a technique to verify adherence to a security policy statically. To do this, we consider a security automaton as a specification, and we generate JML annotations that implement a monitor into the application. We describe this translation and prove preservation of program behaviour, i.e., if monitoring does not reveal a security violation, the generated annotations are respected by the program. The correctness proofs are formalised using the PVS theorem prover. This reveals several subtleties to be considered in the definition of the translation algorithm and in the program requirements.

I started this work while doing an internship at INRIA Sophia Antipolis, France in 2006 under the direction of Marieke Huisman. My contribution to this work was the proposal of using security automata to describe properties and writing and proving a considerable part of the long and complex PVS specification. I presented this work at *12th International Conference on Fundamental Approaches to Software Engineering (FASE 2009)*, part of *ETAPS 2009*, on March 26 2009 in York, UK. This chapter is based on [HT09].

Chapter 4 introduces a framework to reason about the effects of assignments in recursive data structures. We define an operational semantics for a core language based on Bertrand Meyer's ideas for a semantics for the object-oriented language Eiffel [Mey03]. A series of field accesses, e.g., $\mathtt{f}_1 \bullet \mathtt{f}_2 \bullet \ldots \bullet \mathtt{f}_n$, can be seen as a path in the heap. We provide rules that describe how these *multidot* expressions are affected by an assignment. We prove the correctness of a list reversal algorithm using multidot expressions to construct an abstraction of a list. This approach does not require induction and the reasoning about the assignments is encapsulated in the mentioned rules. We also discuss how to use this approach when working with other data structures and how it compares to the inductive approach. The framework, rules and examples are formalised and proven correct using the PVS proof assistant.

This work is based on my master thesis directed by Javier Blanco. I proposed Ken Madlener to join me in working on this topic and I greatly contributed to the PVS formalisation. I presented this work in the *13th Brazilian Symposium on Formal Methods (SBMF 2010)*, on 11 November 2010, in Natal, Brazil. The content of this chapter is based on [TM11].

Part II contains three chapters on resource analysis using size-aware type systems.

Chapter 5 gives an introduction to resource analysis. We overview the most prominent publications in topics such as sized types, amortised cost analysis, automatic complexity analysis, worst case execution time analysis, and others.

Chapter 6 presents a size-aware type system for a first-order functional language with algebraic data types, where types are annotated with polynomials over size variables. We define how to generate typing rules for each data type, provided its user defined size function meets certain requirements. As an example, a program for balancing binary trees is type checked. The type system is shown to be sound with respect to the operational semantics in the class of shapely functions. Type checking is shown to be undecidable, however, decidability for a large subset of programs is guaranteed.

I was the main contributor to the research in this work. I proposed an extension to [SvKvE07a] to cope with algebraic data types and user defined size functions. I presented these ideas at the *9th International Symposium on Trends in Functional Programming (TFP 2008)*, on May 28 2008 in Nijmegen, The Netherlands. This chapter is based on the publication [TSvE09] and its complementary technical report [TSvE08].

Chapter 7. In this chapter we overcome the main restriction of the type system of Chapter 6, i.e., the requirement of shapeliness. This work introduces collected size semantics of strict functional programs over lists. The collected size semantics of a function definition is a multivalued size function that

collects the dependencies between every possible output size and the corresponding input sizes. Such functions annotate types and are defined by conditional rewriting rules generated during type inference. Using collected size semantics we are able to infer non-monotonic lower and upper polynomial bounds for many functional programs. As a feasibility study, we use the procedure to infer lower and upper polynomial size-bounds on typical functions of a list library.

In this chapter we work only with matrix-like structures, however, we have shown that there are ways to relax this restriction [SvET08b].

I contributed to the development of the main ideas in this work, to the soundness proof and its publication. I presented this work at the *20th International Symposium on the Implementation and Application of Functional Languages (IFL 2008)*, on September 10 2008 in Hatfield, UK. This chapter is based on the publication [SvET11], extended with material from the technical report [SvET08a].

There is also an appendix where we include the full proofs of the soundness theorems of Chapters 6 and 7.

The rest of this chapter gives an overview of the background behind the ideas of the following chapters. Part I is concerned with program analysis aided by theorem provers, in particular PVS. Consequently, the next section gives an overview of interactive theorem provers and an introduction to PVS. Part II describes the use of type and effect systems for resource analysis. Type systems, and in particular type and effect systems, are described in Section 3 while Section 4 gives a short motivation for the study of resource analysis.

2 Interactive Theorem Provers

Interactive theorem provers (or theorem provers for short) are software tools that aid in proving logical formulae. We are interested in the use of theorem provers to verify program correctness. To this end, the source code is modelled (usually with abstractions and simplifications) in the language of the theorem prover (called the *proof* or *specification language*) and it is then treated as a mathematical object. The properties one wants to prove about the program are also expressed in the proof language as logical properties of this object. Then, aided by the theorem prover, a user must construct a proof for such properties.

In most theorem provers, proving is goal-driven[3]. The user starts with the formula to be proved as goal. Then she issues a command or tactic that either proves the goal or divides it into (hopefully simpler) subgoals. The system keeps tracks of all subgoals that remain to be proven and ensures that each step is logically correct. When all the subgoals are proven, the whole formula is considered proven.

[3] An exception are declarative proof systems, like Isar [Wen99], that try to emulate the traditional forward way of proving.

This interactivity with the user in helping to construct the proof is the characteristic that distinguishes interactive theorem provers, also known as *proof assistants*, from *model checkers*. In model checking [BK08] one constructs a finite model of the program and the model checker verifies that the formula holds in every state. This is basically done by brute force. Theorem provers are usually more time consuming for the user, but they are also more expressive. Furthermore, theorem provers do not suffer from the combinatorial blow up of the state-space that affects model checkers, commonly know as the *state explosion problem*.

Theorem provers are very useful in the setting of proving program correctness because the verifications tend to be very large with many uninteresting and similar proofs. The tool helps with all the bookkeeping, ensuring that nothing is forgotten and that there is no oversight. Proving a long specification is an arduous work, but once the proofs have been written they can be checked within minutes. For example, the proofs developed for Chapter 3 took months to write, but they can be checked in less than an hour with a modern computer. *Proof carrying code* (PCC) [Nec97] takes advantage of this fact. In PCC, a host system like a mobile phone can quickly verify properties about an untrusted application via a formal proof that accompanies the application's executable code. Then, the host system can determine whether the application is safe to execute, without having to monitor it at run-time.

Among the most well-known general-purpose theorem provers we can name PVS [OSRS01], Isabelle [Wen10], Coq [Tea10] and ACL2 [KM11, SBB+01]. For a comprehensive (but not up-to-date) listing of reasoning systems, the reader can check the *Database of Existing Mechanised Reasoning Systems* [KT]. In Chapter 3 of her PhD thesis [Hui01], Marieke Huisman describes both PVS and Isabelle in detail and an ideal system taking the best of each tool. PVS has been our tool of choice and is described in more detail below.

Some theorem provers are specialised to a particular programming language like KeY [ABHS07] for Java or Sparkle [dMvEP08, dM09] for Clean [PvE98]. Having a restricted domain allows these tools to be more specific and to integrate better in the program development environment. There are also projects that model practical languages using theorem provers. This can be used, for example, to prove soundness of the type system, but also to formally reason about programs written in that language. Examples are the LOOP compiler [vdBJ01] for sequential Java and JML developed using PVS, and Compcert, a compiler for a large subset of the C verified with Coq.

There is abundant research on the topic of automated theorem proving, which is in fact a subfield of *automated reasoning*, see for instance [WOLB92, Fit96]. However, automated theorem proving is not a subject of this thesis; we have merely used theorem provers as a tool to develop and validate our specifications.

PVS

The *Prototype Verification System* (PVS) [ORR+96, OSRS01] is an interactive environment developed at the Stanford Research Institute (SRI) for writing formal specifications and mechanically checking proofs. It is composed by a proof language, the theorem prover itself and a graphical interface. Each component is described below.

The specification language The specification language of PVS is based on classical, typed higher-order logic. A higher-order logic, unlike a first-order one, allows quantification over propositions and predicates.

We describe the main characteristics of the language by some examples: a simple axiomatic formalisation of stacks, an inductive definition of binary trees and a coinductive definition of streams. In general, inductive definitions are preferred to axiomatic ones.

```
stacks[T: TYPE+] : THEORY
BEGIN
  stack : TYPE+
  s : VAR stack
  empty : stack
  nonemptystack?(s) : bool = s ≠ empty

  push : [T, stack → (nonemptystack?)]
  pop  : [(nonemptystack?) → stack]
  top  : [(nonemptystack?) → T]

  x, y : VAR T

  push_top_pop : AXIOM nonemptystack?(s) ⇒ push(top(s), pop(s)) = s
  pop_push : AXIOM pop(push(x, s)) = s
  top_push : AXIOM top(push(x, s)) = x
  pop2push2: THEOREM pop(pop(push(x, push(y, s)))) = s
END stacks

binary_tree[T : TYPE] : DATATYPE
BEGIN
  leaf : leaf?
  node(v : T, left, right : binary_tree) : node?
END binary_tree

stream[T : TYPE] : CODATATYPE
BEGIN
  cons(first : T, rest : stream) : cons?
END stream
```

PVS specifications are organised into parametrised theories that may contain assumptions, definitions, axioms, and theorems. In the example, the three theories are parametrised by a type `T`.

The base types include uninterpreted types that may be introduced by the user, like `stack`, and built-in types such as the booleans, integers and reals. Type constructors include functions, sets, tuples, records, enumerations, inductively defined abstract data types like `binary_tree`, and coinductively defined abstract data types such as `stream`.

In the definition of `stacks`, `s` is a *variable* `stack` while `empty` is a *constant* `stack`. The variables `x` and `y` are of type `T`, hence `T` is declared with the keyword `TYPE+`, indicating that it must be a non-empty type. However, this is not the case for `binary_tree` and `stream`, because they do not instantiate the type parameter.

Formulas are terms of type `bool`. We shall use the standard notation for connectives ($\wedge, \vee, \Rightarrow, \neg$), and for quantifiers ($\forall$, $\exists$). There is a conditional term IF φ THEN M ELSE N, for terms M and N of the same type.

Two outstanding characteristics of PVS are the use of *dependent types* and *predicate subtypes*. Dependent types are types parametrised by values. A predicate subtype of a type `T` is a new type defined as the elements of `T` that satisfy a given predicate. Dependent types and predicate subtypes work very well together, as we can see in the following definition of vectors:

```
vector(n : nat) : TYPE = {l : list[T] | length(l) = n}
```

The type `vector` is parametrised by a natural number `n`, which is used in the predicate to restrict the length of instances of `vector(n)` to `n`.

As in most theorem provers, in PVS functions must be total. However, predicate subtyping makes it easy to constrain the domain, like in the definitions of `pop` and `top`, which are defined only for non-empty stacks. Note that the predicate `nonemptystack?` is being used as a type in the definition of these functions. In PVS any predicate may be used as a type simply by putting parentheses around it. Since a type can be defined by any predicate, type checking in PVS is undecidable. When the system cannot syntactically determine that an element belongs to a subtype, it generates a *type correctness condition* (TCC). These TCCs are proof obligations, but most of them are discharged automatically by the theorem prover.
In our example we get two TCCs:

```
% Existence TCC generated for push: [T, stack → (nonemptystack?)]
push_TCC1: OBLIGATION
  ∃ (x: [[T, stack] → (nonemptystack?)]): TRUE

% Subtype TCC generated for pop(push(x, push(y, s)))
% expected type (nonemptystack?)
pop2push2_TCC1: OBLIGATION
  ∀ (s: stack, x, y: T): nonemptystack?(pop(push(x, push(y, s))))
```

Let us start looking at the second type correctness condition. The function `pop` expects an element of type (`nonemptystack?`) and returns a value of type `stack`. This works fine for the inner `pop` because it is applied to `push`, which returns an element of type (`nonemptystack?`); but the outer occurrence of `pop` cannot be seen to be type correct by such syntactic means, hence a TCC is generated. It is easily proved using the `pop_push` axiom. The first TCC requires us to prove that the type [`T, stack`] $\rightarrow$ (`nonemptystack?`) is non-empty (because we are instantiating it). At the point of definition of `push` we have only said that `stack` is a non-empty type containing `empty`. Thus `stack` may very well be the type whose only inhabitant is `empty`; in that case a function like `push` would not exists. This shows how easy is to overlook details in a formal specification (especially in an axiomatic one). One way to solve this is to add another axiom before the definition of `push`, saying that the type (`nonemptystack?`) is indeed non-empty:

```
nonemptystack_nonempty : AXIOM ∃ (x: (nonemptystack?)): TRUE
```

Definitions are guaranteed to be conservative extensions. To ensure this, recursive function definitions require a well-founded measure. Inductively-defined relations are also supported. For each inductive data type a number of theories are automatically generated. These theories contain, for instance, axioms about extensionality, an induction principle and *fold* and *map* function definitions for the data type.

PVS expressions provide the usual arithmetic and logical operators, function application, lambda abstraction, and quantifiers, with a natural syntax. Names may be freely (statically) overloaded, i.e., declarations with the same name are allowed as long as they have different types. An extensive prelude of built-in theories provides hundreds of useful definitions and lemmas; user-contributed libraries provide many more.

The prover The PVS theorem prover provides a collection of powerful primitive inference procedures that are applied interactively under user guidance within a sequent calculus framework.

Just to give a flavour of how proving works in PVS, we show how to prove the theorem `pop2push2` of our example. We start with the sequent

```
  |-------
{1}   ∀ (s: stack, x, y: T): pop(pop(push(x, push(y, s)))) = s
```

The first thing we want to do is to get rid of the universal quantifier by instantiating the variables. This is known as *skolemisation*. For that we enter the command (`skolem`)[4], which transforms the sequent into

```
  |-------
{1}   pop(pop(push(x!1, push(y!1, s!1)))) = s!1
```

[4] Proof commands are written in lisp notation, i.e., they are *s-expressions*. Thus, each command is surrounded by parentheses.

We now want to apply the `pop_push` axiom to cancel out the inner application of these functions. For that we issue the command `(use "pop_push")` which tries to find the best instantiation of `pop_push` for the task at hand. We get the following sequent

```
{-1}  pop(push(x!1, push(y!1, s!1))) = push(y!1, s!1)
  |-------
[1]   pop(pop(push(x!1, push(y!1, s!1)))) = s!1
```

The new formula is added before the line dividing the sequent, meaning that it is a premise. Each formula in the sequent is assigned a number that can be used to refer to it. Premises have negative numbers while conclusions have positive ones. The formulae that were affected by the previous command surround its number in curly braces and the ones that did not change use square braces. Now, with `(replace -1 1)` we replace the left-hand side of the premise by its right-hand side in the conclusion:

```
[-1]  pop(push(x!1, push(y!1, s!1))) = push(y!1, s!1)
  |-------
{1}   pop(push(y!1, s!1)) = s!1
```

Then we just need to apply `pop_push` once more. If we issue `(use "pop_push")` again, we get the premise repeated. So we instantiate the axiom manually with `(lemma "pop_push" ("x" "y!1" "s" "s!1"))`, where we are saying that `x` is now `y!1` and `s` is `s!1`. This concludes the proof:

```
{-1}  pop(push(y!1, s!1)) = s!1
[-2]  pop(push(x!1, push(y!1, s!1))) = push(y!1, s!1)
  |-------
[1]   pop(push(y!1, s!1)) = s!1

which is trivially true.
Q.E.D.
```

The primitive inference methods include propositional and quantifier rules, induction, rewriting, simplification using decision procedures for equality and linear arithmetic, data and predicate abstraction, and symbolic model checking. The implementations of these primitive inferences are optimised for large proofs: for example, propositional simplification uses binary decision diagrams (BDDs), and auto-rewrites are cached for efficiency. User-defined procedures can combine these primitive inferences to yield higher-level proof strategies. However, to define tactics, the user needs a good understanding of the internals of PVS and Lisp. The PVS specification language and the prover cooperate in such a way that the type information associated with a term is exploited by the inference mechanisms, and conversely, the automation in the prover is helpful in automatically discharging TCCs.

PVS has been extended in several ways:

- with BDD-based decision procedure for the relational μ-calculus, providing an experimental integration between theorem proving and CTL model checking [ORR+96]. CTL stands for Computation Tree Logic, a logic used in model checkers to express properties,
- with the YICES [dMD06] Satisfiability Modulo Theories (SMT) solver as an endgame prover and an infinite-state bounded model checker,
- with the PVSIO [Muñ05] framework for evaluating ground PVS expressions,
- and with a random testing capability that can be used during proofs [Owr06].

The user interface PVS uses EMACS to provide an integrated interface to its specification language and prover. There is also a batch mode used to run proofs automatically.

Proofs are written in a special EMACS proof mode. Proofs yield scripts that can be edited and rerun. These scripts are stored separately from the specification in `.prf` files. A theorem can have more than one proof. Proof trees and theory hierarchies can be displayed graphically using TCL/TK, see Figure 2. This is helpful to see the history of a branch and which subgoals remain to be proven.

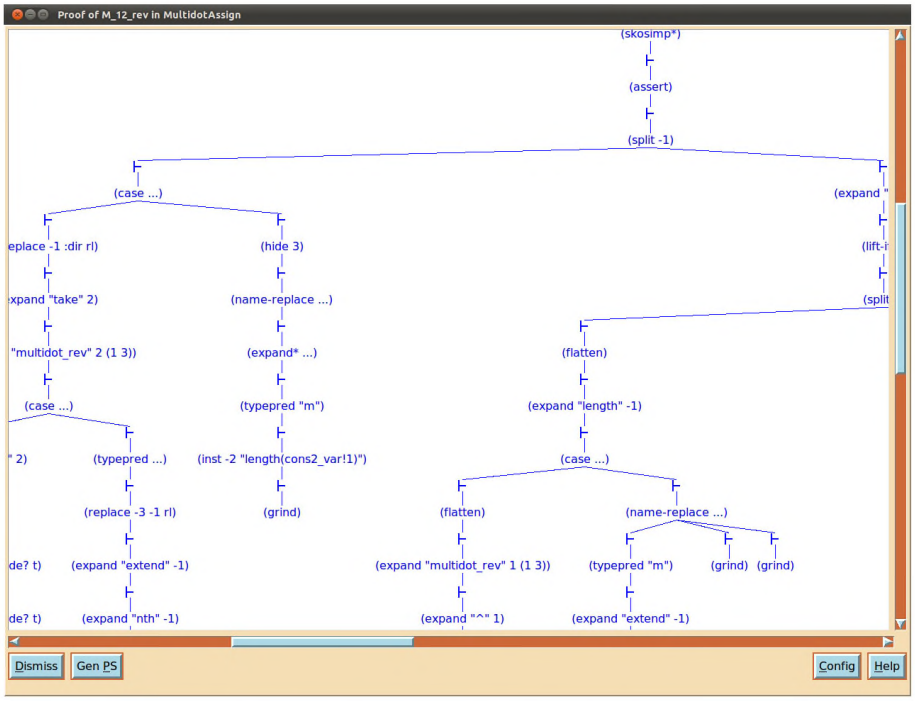


Fig. 1. Example of a TCL/TK proof tree.

Extensive help, status-reporting and browsing tools are available, as well as the ability to generate specifications typeset in LaTeX with user-defined notation.

3 Type Systems

Benjamin Pierce defines a type system as a "a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute" [Pie04]. A type system associates types with each computed value. By examining the source code of a program, a type system attempts to guarantee that operations expecting a certain kind of value are not used with values for which that operation makes no sense. For instance, a simple type system can ensure that only numeric values are used in an addition. If a boolean value can make its way and end up as an argument, the type system will detect that and will output an error.

Type systems can also be used to guarantee more complex properties like that there are no null-dereferences [MPPD08], or out-of-bounds array accesses [XP98] or that no incompatible units are used in a scientific calculation [Ken94]. They can also be used to do alias analysis [SWM00, Kon03] or, as we describe in Chapters 6 and 7, to do resource analysis.

The process of verifying the constraints imposed by types – *type checking* – can occur either at compile-time or at run-time. The former is called *static* and the latter *dynamic*. In this section we consider only static type checking, however, in Chapter 7, the inference procedure has a dynamic aspect.

This section gives a short introduction to type systems. In 3.1 we describe the benefits of using type systems and in 3.2 we discuss type and effect systems, a particular class of type systems used in Part II.

3.1 Advantages of Type Systems

The use of type systems is convenient for many reasons:

Guaranteed properties The most obvious benefit of static type checking is that of static analysis in general: early error detection. Type checking can expose a broad range of errors, not only simple mistakes like forgetting to convert a string to a number when asking if it is positive, but also possibly conceptual errors like using the wrong data structures. As Robin Milner said more that 30 years ago [Mil78]: *well-typed programs cannot "go wrong".*

Type systems can also be used to prove properties like termination or to prove that the execution meets some resource bounds, as we will see in Chapter 5.

Abstraction Another key advantage of types is that they allow the programmer to think in terms of entities instead of thinking in the details of the representation. For instance, it is easier to think of a date as a structure that somehow stores the day, the month and the year rather than having to remember if in the memory layout the day is stored before or after the month.

Information hiding, the principle of providing an interface to interact with some data to protect the rest of the program from changes in its representation, is easier to achieve in a system with an expressive type system.

Maintenance Different types can have the same representation, but their distinction makes it explicit that they are used for different purposes and makes it easier to find the places where they are used. For example, a date can be internally represented as integer that counts the number of days from say 1980. It is much easier to find all the places where dates are used if they are an instance of a type date rather than an integer.

Documentation Types are usually named after the entity or abstraction they represent. Meaningful type names can greatly help in understanding the purpose of a program. Furthermore, unlike comments, this form of documentation cannot become outdated.

Optimisation Static type checking obviates the need for dynamic checks that would be needed to guarantee safety and their associated cost. It can also provide useful information to the compiler, which may, for instance, be able to use more efficient machine instructions.

3.2 Type and Effect Systems

Type and effect systems are program analyses that extend basic types with annotations describing properties of values or evaluations. A good introduction to this topic can be found in Chapter 5 of [NNH99].

Analyses based on effects were first introduced to control the combination of imperative features with functional languages [Luc87, GL86, TJ94]; in this setting, effects are abstract descriptions of impure side effects occurring during evaluation, e.g., accesses to imperative references or input-output actions. Other uses of type and effects analysis include exception tracking, inferring region annotations [TJ92, TB98], analysing communication in concurrent systems [ANN99] and predicting execution costs [DJG92, RG94, HP99].

As a simple example consider *sharing analysis* (see, e.g., [TWM95, Wan02]). Sharing analysis is a static analysis that aims at determining which objects are used at most once. Hage et al. [HHM07] formalise the relationship with the similar *uniqueness analysis* [PvE93, BS93, BS96, dVRM08]. It is primarily targeted at languages with lazy, i.e., call-by-need evaluation, such as Haskell [Jon02] and Clean [PvE98]. Call-by-need evaluation is usually implemented by means of updatable nodes in a graph [vE88]. For instance, in the program

$$\textbf{let } x = 2 + 3 \textbf{ in } x + x,$$

the variable x is represented by a node that initially contains the subterm $2 + 3$. The first time the value of x is required, it is calculated and this node is updated with the result. The second time its value is required, it can be immediately

retrieved so the evaluation of $2 + 3$ is effectively shared between the occurrences of x in the body of the local definition. However, in the program

$$\textbf{let } y = 2 + 3 \textbf{ in } 2 * y,$$

the value of the variable y is required only once and, hence, its evaluation will not be shared. Therefore, there is no need to update the node for y, and in fact it makes little sense. In general, it is unnecessary to update a node if its value is used at most once. Instead, after it has produced its value, such a node can be garbage collected.

A sharing analysis typically classifies terms into two groups: those that are guaranteed to be used at most once and those that may be used more than once. Type-based sharing analyses record these classifications in type derivations. For instance, as done in [HHM07], such an analysis can produce a typing like

$$x :^{\omega} \textbf{Int}.$$

Here, ω indicates that the evaluation of x may be shared, while

$$y :^{1} \textbf{Int},$$

indicates that y is used at most once. In type-based program analysis, annotations such as 1 and ω are often referred to as *effects*. In the specific context of usage analysis, they are called usage effects or usage annotations.

Effects can also appear within function types. Consider, for instance, the function definition

$$double\ x = 2 * x$$

and the typing

$$double :^{\omega} \textbf{Int}^{1} \to \textbf{Int}^{\omega}.$$

The domain and codomain of the function type are annotated with usage effects. The effect 1 on the domain indicates that *double* uses its argument at most once; the effect ω on the codomain indicates that results produced by double may be used more than once. The remaining ω ranges over the whole typing and expresses that the function itself may be shared.

4 Resource Analysis

Software is increasingly used in systems that are not general-purpose computers and whose primary role is to interact with the environment, e.g., in a microprocessor-controlled washing machine, in the control of an industrial robot or in an aircraft flight controller. Such software is generically designated as embedded, to signify that it is part of a larger engineering system which it must support [BW01].

Embedded software must satisfy both functionality requirements (that responses are logically correct) and resource guarantees (that responses are computed within fixed bounds on time, dynamic memory or energy consumption).

In order to meet the latter non-functional requirements while making efficient use of available hardware, embedded systems were traditionally programmed in low-level assembly languages.

The increasing complexity of applications means that it is no longer cost-effective to develop embedded systems solely in low-level languages like assembly. Today's embedded systems are typically developed in higher-level languages such as C, C++, Ada or even Java with only a very limited part in assembly code. The use of higher-level languages eases the detection and correction of errors, facilitates portability and re-use of code and generally increases the speed of development. The price of such facilities is a loss of predictability of the time and space usage of programs.

For applications that are not critical, the loss of predictability can be mitigated during testing, e.g., by profiling time and space usage. Testing, however, is very time-consuming and the results can be invalidated by even minor code changes during development. Moreover, as pointed out by Dijkstra [DDH72] almost four decades ago, testing can show the presence but never the absence of errors.

The current software development practice for high-integrity systems like systems for aerospace, rail or government security, is to prohibit language features that may lead to unpredictable resource usage. A good example is SPARK, which was designed with verifiability as the primary design goal. SPARK is a subset of the Ada language that excludes recursion and dynamic memory allocation [FA08]. In other languages like Java Card, memory must be allocated in an initialization phase after which no memory allocation can occur. However, this is done at the loss of useful programming techniques and abstractions. For example, Stankovic [Sta88] points out that dynamic memory allocation is essential for the next-generation of embedded systems. As embedded applications become more complex, there is an increasing interest in techniques that combine programming languages, formal methods and implementations that take advantage of high-level abstractions while at the same time guaranteeing safety properties of the executed code.

In Chapter 5 we give a survey that describes the most influential works in this field.

Part I

Proving Program Properties Using Formal Models

Chapter 2

Preemption Abstraction: A Lightweight Approach to Modelling Concurrency

Revised version of *Preemption Abstraction: A Lightweight Approach to Modelling Concurrency.*
In Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2009). Eindhoven, The Netherlands, November 2-3, 2009.

Preemption Abstraction

A Lightweight Approach to Modelling Concurrency

Erik Schierboom[3], Alejandro Tamalet[1⋆], Hendrik Tews[1⋆⋆], Marko van Eekelen[12], and Sjaak Smetsers[1]

[1] Institute for Computing and Information Sciences (iCIS), Radboud University Nijmegen, The Netherlands
[2] School of Computer Science, Open University, The Netherlands
[3] BliXem Internet Services

Abstract. This chapter presents the *preemption abstraction*, an abstraction technique for lightweight verification of one sequential component of a concurrent system. Thereby, different components of the system are permitted to interfere with each other. The preemption abstraction yields a sequential abstract system that can easily be described in the higher-order logic of a theorem prover. One can therefore avoid the cumbersome and costly reasoning about all possible interleavings of state changes of each system component. The preemption abstraction is best suited for components that use preemption points, that is, where the concurrently running environment can only interfere at a limited number of points. The preemption abstraction has been used to model the IPC subsystem of the Fiasco microkernel. We proved two relevant properties of the model: that after a send operation the sender wakes up the receiver and that it releases the lock on the receiver. On the attempt to prove a third property, namely that the assertions in the code are always valid, we discovered a bug that could potentially crash the whole system.

1 Introduction

In this chapter we focus on the verification of the following kind of systems: a component $\mathcal{C}$ is running in a concurrent environment $\mathcal{E}$, where $\mathcal{E}$ interferes asynchronously with the component $\mathcal{C}$ by, for instance, changing some state variables of $\mathcal{C}$. The goal is to prove functional properties about the component, regardless of how the environment behaves.

This kind of problem appears for instance in operating-system verification. Every recent operating system permits several threads of execution running in quasi-parallel, even on a system with only one processor core. Typically each such thread might invoke any operating-system call. Nevertheless, the effects the different threads might have on each other are relatively limited. For the verification of the operating system, it is therefore often sufficient to consider only

⋆ Sponsored by the Netherlands Organisation for Scientific Research grant 612.063.511.
⋆⋆ Supported by the European Union through PASR grant 104600.

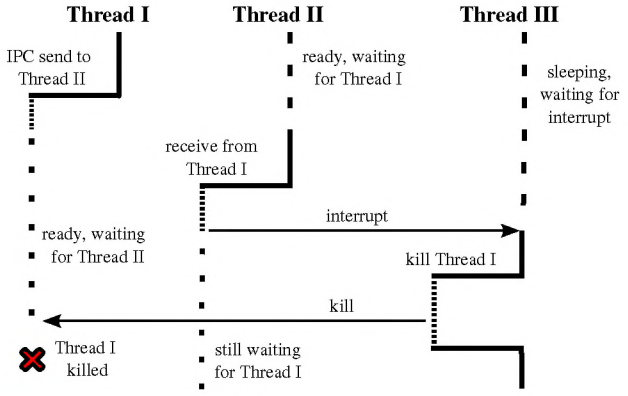


Fig. 1. Environment (thread II + III) asynchronously interfering with thread I. The zigzags in the lines represent system calls: the threads are executing user code in the solid lines and operating-system code in the dotted lines. Dashed lines and separated dots indicate that the thread is not scheduled.

one thread of execution, and to model all the threads that can asynchronously affect the given thread as some kind of environment.

As an example, Figure 1 shows three threads. Initially, thread I and thread II want to exchange a message via inter-process communication (IPC), while thread III is sleeping. Thread II and thread III can be considered as the environment of thread I, that is, they can asynchronously affect thread I. When thread I performs a system call in order to send a message to thread II, the environment could react in several ways (where only the last one is displayed in Figure 1):

- The environment could do nothing, corresponding to a situation where thread II never performs the system call necessary to receive from thread I.
- The environment could engage in IPC with thread I, corresponding to a situation where thread II successfully receives the message from thread I.
- The environment kills thread I, as displayed in Figure 1. Here thread II starts the system call to receive from thread I, but then an external interrupt wakes up thread III. Thread III immediately gets scheduled (for instance because it has a higher priority) and kills thread I.

It is important to notice here that the number of different effects that the environment can have on thread I, is rather limited. Although every thread runs arbitrary user code, there is only a limited number of system calls and only few of them can have an effect on other threads.

Only few operating-system kernels are fully interruptible, meaning that rescheduling of a different thread can occur at every point in every kernel procedure. Maintaining consistency of kernel data structures for a fully interruptible kernel is difficult, therefore many kernels disable rescheduling or even interrupts over large portions of the kernel. When real-time properties are a concern, a kernel design with *preemption points* is sometimes used. In this design, interrupts

(and therefore rescheduling) are generally disabled, except at well-defined points —the preemption points. Pending interrupts are then delivered only at these points. Kernel data structures are synchronised before any preemption point so that rescheduling a different thread (which might engage in different kernel activities) can be done without danger of corruption.

In this article we describe and use the *preemption abstraction*, an abstraction technique tailored for this kind of systems. The technique has been developed for creating and verifying models in the higher-order logic of an interactive theorem prover. The preemption abstraction is equally well applicable in a model-checking environment Because model checkers provide built-in support for analysing parallel systems, the preemption abstraction will not be as useful as in theorem proving for writing the model, however, it will contribute to reduce the state space. We used the abstraction technique in the modelling and verification of the inter-process communication (IPC) facilities of the microkernel Fiasco [HH01, Hoh98, HP01]. Our verification attempt identified one programming error, although the part of the IPC subsystem that was modelled was thoroughly tested and in daily use. The bug could only be triggered when a specific interrupt occurred precisely in a very short time frame during the execution of the IPC system call. It was therefore so unlikely to trigger the bug that it could have stayed unidentified for decades.

This chapter is organised as follows. The next section describes the preemption abstraction while Section 3 describes Fiasco, in particular its IPC subsystem. In Section 4 the PVS model is discussed, with emphasis on the application of the preemption abstraction. Section 5 comments on the properties that were verified and on the programming error that was found. In Section 6 we evaluate the case study and give pointers to future work. Finally, Section 7 discusses related work and Section 8 draws conclusions.

2 The Preemption Abstraction

Consider a parallel system $\mathcal{S}$, as exemplified in the introduction, with the following properties. $\mathcal{S}$ consists of an arbitrary number of threads and each thread consists of a sequence of atomic blocks. Between each two atomic blocks there is a preemption point, in which no computations and state changes are performed (in practice a preemption point consist of one or two *NOP* instructions). For each atomic block, each thread acquires a global lock, which is released during the preemption points. Thus, a computation of the whole system consists of one sequential interleaving of all the atomic blocks. Apart from the sequential interleaving, the threads may interfere in arbitrary ways, for instance, a thread t_1 may change the state of another thread t_2. Because of the sequential interleaving, t_2 is of course waiting in a preemption point when t_1 changes its state.

The system $\mathcal{S}$ is parallel in the sense that for a complete description of its behaviour it is necessary to consider all possible interleavings of the atomic blocks of all threads in $\mathcal{S}$.

The preemption abstraction focuses on one selected thread, say t. All other threads are considered as the environment of t. When t is waiting for the global lock in a preemption point, any thread from the environment can change the state of t. All such potential changes are collected in the set of side effects SE. For real systems this set would typically have a small finite cardinality, but the correctness of the abstraction does not depend on that. In this work we assume that the events in SE are independent, however, the abstraction could still be applied if such dependencies are made explicit when we ask for the possible events that can occur at a certain preemption point. For example, if the events e_1 and e_2 are such that e_2 must be preceded by e_1, the list of possible effects should no include e_2 if e_1 has not occurred yet.

In the following we consider (finite) lists of side effects taken from SE. To execute such a list means to perform all the side effects in the order of the list. Note that one particular side effect can occur multiple times in such a list.

The preemption abstraction $\mathcal{A}$ of the system $\mathcal{S}$ consists *only* of the thread t with the following changes:

- The *preemption-point function* is substituted for all preemption points in t.
- The preemption-point function chooses a list of side effects and executes it.
- The global lock, its acquisition and release are abstracted away.

In the preemption abstraction all the other threads of $\mathcal{S}$ that form the environment of t are condensed into the preemption-point function. The way the side effects are chosen depends on the particular system; if there are no dependencies among them then they can be chosen nondeterministically. Note that the chosen list can be the empty list.

The preemption abstraction $\mathcal{A}$ is a sequential model of $\mathcal{S}$ that faithfully models the behaviour of the thread t. Under the assumption that there are no dependencies between the threads the abstraction suffices to prove arbitrary (functional) properties of t that can be proved in $\mathcal{S}$. Since it takes the point of view of a single thread, the abstraction cannot be used to prove properties about cooperating threads. The abstraction is sound in the sense that every property proved for t in $\mathcal{A}$ also holds in $\mathcal{S}$. The soundness crucially depends on the completeness of the set of side effects SE.

In a single-core architecture there is no concern with shared data because a preemption point runs without interrupt, and thus no other thread can change the data. In a multi-core architecture we can still use the preemption abstraction if a thread accesses shared data only at the beginning of a preemption point, for instance to check the state of another thread, but during the preemption point it only accesses local data. As will be discussed in Section 6, this is not the case for many programs. Although the preemption abstraction could still be used in principle by taking the atomic blocks at the memory level, it would not be beneficial.

The main advantage of the preemption abstraction $\mathcal{A}$ is that it is a sequential model, consisting of only one thread. For a description of its behaviour one does not have to consider different interleavings of atomic blocks. The abstraction

$\mathcal{A}$ can therefore be conveniently described as a functional model in the higher-order logic of an interactive theorem prover such as PVS [ORR+96]. In contrast, modelling the behaviour of $\mathcal{S}$ with all possible interleavings of its threads in higher-order logic would be a major hassle. The preemption abstraction is therefore absolutely necessary in order to verify nontrivial systems $\mathcal{S}$ in an interactive theorem prover.

In this chapter we use the preemption abstraction in the context of operating-system verification. We are currently only interested in functional correctness properties of single operating-system calls. Therefore we abstract from the concrete user-mode programs and consider only a system that runs a *most general user-mode program* on an unspecified number of threads. This most general user-mode program continuously performs random operating-system calls with randomly chosen arguments. All threads in our system $\mathcal{S}$ execute the same (most general) program. As a result the set of side effects SE is also most general: it will contain all possible side effects of all system calls. After the set of side effects SE has been determined we also abstract away the most general user-mode program and henceforth consider only the system calls of interest.

3 Interprocess Communication in Fiasco

The Fiasco microkernel belongs to the L4 microkernel family. It has been developed since 1998 at TU Dresden, Germany. It is mainly written in C++ with some inline assembly and assembly shortcuts for the system calls whose performance is critical. In a microkernel-based system many operating-system services are implemented as separate modules, which are running as normal application programs. Therefore inter-process communication (IPC) is often the bottleneck of microkernel based systems. With very stringent optimisations, the L4 microkernel interface and some of its implementations remedied this problem, achieving performance within 5% of traditionally designed systems [HHW98]. The L4 family (and other microkernels) is therefore sometimes referred to as a microkernel of the second generation.

The Fiasco microkernel implements processes, threads, address spaces, interprocess communication and delegation of memory resources. The only device that the kernel controls itself is the interrupt controller. Drivers for all other devices, such as hard disks, graphic cards and keyboards run outside of the kernel as normal application programs.

IPC will play an important role for this chapter, so let us elaborate a little bit on it. IPC in the L4 interface is optimised for the common case of client-server communication. There is just one system call for IPC, whose precise behaviour can be modified via certain parameters. IPC in Fiasco is always synchronous, that is, sender and receiver must perform a rendezvous. If either the sender or the receiver is not ready, the other party blocks. In general the IPC system call always performs a send operation followed by a receive. Both the send and the receive operation are optional and can be disabled via parameters to obtain a

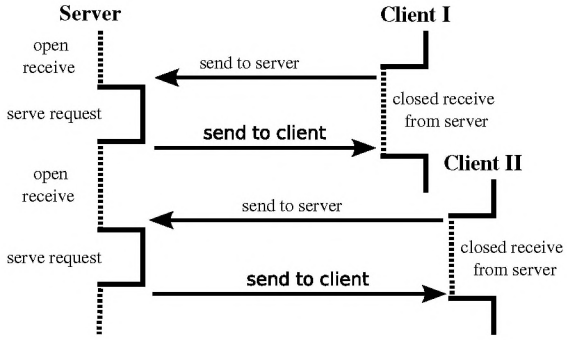


Fig. 2. Typical communication pattern for applications running on an L4 microkernel. As before solid lines indicate user code and dotted lines indicate operating-system code.

send-only or receive-only IPC system call. If the send operation is enabled it always sends to a specified destination thread. The receive operation can be either *open* or *closed*. In an open receive any IPC partner is accepted, while in a closed receive only messages from one specified thread are accepted. Both the send and the receive operation always transfer two registers plus, optionally, some memory contents. If some memory is copied it is called *long IPC*, otherwise *short IPC*. Typically short IPC prevails and shared memory is used for bulk data transfer. The time the IPC system call blocks in either the send or the receive operation can be controlled via timeout parameters. As special cases the timeout can be zero (abort IPC if the partner is not ready) or infinite (no timeout).

Figure 2 shows how the IPC operation is exploited in client-server communication. At the beginning the server blocks with infinite timeout in an open receive until client I starts a complete IPC call. This call consists of a send operation and a closed receive, both with the server as IPC partner. When the send operation from client I to the server is complete, the server finishes its IPC system call and starts working on the client request. Meanwhile, client I blocks in a closed receive (typically with infinite timeout) until the server answers.

When the server finishes working on the request, it starts a new complete IPC system call. In the send operation it sends its answer back to client I. Client I thereby finishes its IPC system call and continues normal computation. After sending the answer, the server blocks in an open receive waiting for the next client. The server can thus be programmed in a loop with one IPC system call as last statement of the loop. At server boot time, just before entering its main loop, the server does an open receive without send operation.

In Fiasco IPC is implemented such that the sender is the active part. That is, the sending IPC partner performs the necessary locking and copies the message. The receiving IPC partner simply waits until some sender finished its job.[4]

In Fiasco, thread ID's are 64-bit numbers. They are used to denote potential senders and receivers. There are two special thread ID's: the invalid thread ID, sometimes referred to as null-thread ID, and the nil-thread ID. The nil-thread ID can for instance be used in a closed receive with some timeout. As effect the thread will sleep until the timeout elapses.

A major performance cost of IPC are address-space switches, which are necessary when, as in Figure 2, the client blocks and computation continues in the server in a different address space. There is an important optimisation to minimise the IPC related address-space switches: The kernel data structures of the client are prepared for the receive operation *before* switching to the server. This way the kernel can immediately start copying the message to the client when the server replies. Because of this optimisation, the IPC procedure in the kernel prepares the receive part before it starts the send part—although the actual receive only follows the send operation.

4 The Model

This section describes our model of Fiasco's inter-process communication with special emphasis on the abstraction described in Section 2.

For the formalisation we chose the theorem proving approach and, in particular, we used the PVS theorem prover (see Section 2 of the introduction to this thesis). Section 7 describes other works that used the model checking approach to model the same subsystem.

The code that had to be modelled was written in a small subset of C++: mainly assignments, conditionals and method calls. We reduced it even more by abstracting most loops and splitting functions with side effects into a state transformer plus a pure function that returns a value. This resulted in a shallow embedding of the C++ sources in PVS. However, this approach is not always feasible; to be able to model arbitrary code, a complete semantics of the language is needed. This is the path taken by the VFiasco project (see section on related work) for a subset of C++ and by the LOOP project for sequential Java programs [Hui01].

The key aspect of the formalisation is the treatment of parallelism. To avoid defining an interleaving semantics, we used the preemption abstraction described before. The set of possible side effects that can occur at a preemption point was identified by code inspection. Then a preemption point was modelled as a (non-deterministic) function that executes an arbitrary list of these pre-established actions. Hence, we made an abstract formalisation of concurrency by modelling

[4] An exception are interrupts that are mapped into an IPC message to the thread that registered for that interrupt. In this case the receiving thread is active. However, interrupt IPC is not considered throughout this chapter.

the *effects* that the environment can have on a running thread, avoiding giving a full formalisation of the environment.

4.1 Key Abstractions

In a real system executing on the Fiasco microkernel, many threads can run in parallel, and each one can start an IPC system call. Therefore, the IPC code in the kernel potentially runs in parallel with itself many times. In order to obtain a sequential model that can be easily described in PVS, we applied the preemption abstraction as explained in Section 2.

As the first step we identified the set of side effects SE. When a thread t performs an IPC operation other threads can modify the state of t in the following way:

1. The thread t can be killed;
2. A timeout can occur meaning that t should not wait any longer for an IPC partner to become ready;
3. The IPC operation of t can be cancelled;
4. A receiver can become ready, meaning that t can proceed with the send part of the IPC.

The side effects are modelled in PVS with the type `PreemptionAction` and the function `doPreemptionAction`, as we will explain in Section 4.2 below.

As a second step we focused on the IPC code of just one thread, ignoring the rest of the system. The preemption points are replaced by the preemption-point function, which is formalised in PVS by `preemptionPoint`, see Section 4.2 below. Note, that after applying the preemption abstraction there is no scheduler left in the model. The only effect the scheduler could have is that our thread t remains for a longer time in some preemption point and that therefore some more side effects accumulate.

Independently from the preemption abstraction, we decided to focus on the core functionality of IPC. We model only short IPC between real threads. Note that we do model timeouts in an abstract way without any notion of time in the model: A timeout side effect can occur in any preemption point.

4.2 PVS Specification

In PVS a theory is a module that encapsulates definitions and properties. It provides a means to hierarchically decompose a specification. Our work is composed of several theories with simple dependencies among them. The theories `state` and `ipc` contain the model and will be discussed in this section. For reasons of clarity, we will restrict ourselves to some relevant, slightly simplified extracts of our model. The complete specification can be obtained via `http://cs.ru.nl/~marko/research/phds/tamalet/`.

We define `ThreadPointer` as an uninterpreted type, which essentially represents an arbitrary set. This set should have at least two elements, which will

be enforced by an axiom. We say that `null` is a `ThreadPointer`, and declare `NonNullTP` as the set of non-null thread pointers. The `nil_thread_ptr` constant points to the special nil thread (see Section 3), used to encode send-only or receive-only IPCs. The following PVS extract formalise these points.

```
ThreadPointer: TYPE
not_empty_or_single: AXIOM ∃ (tp1,tp2: ThreadPointer): tp1 ≠ tp2
null: ThreadPointer
NonNullTP: TYPE = { tp:ThreadPointer | tp ≠ null }
nil_thread_ptr: NonNullTP
```

Fiasco stores the status of a thread in a bit vector. We have represented the flags of this vector that are relevant to our model as a record with boolean fields. A complete description of the status flags can be found in [Hoh02].

```
ThreadStatus: TYPE = [#
  ready, cancel, dead, busy, invalid,
  polling, receiving, ipc_in_progress,
  send_in_progress, transfer_in_progress: bool
#]
```

Each thread is composed of a `status`, a `partner` to engage with in IPC and a list of senders (named `senders_waiting`) containing the senders that are queued if the receiver is busy. As explained in Section 3, the sender is the active part in an IPC, and one of the actions a sender performs is locking the receiver. In our abstract model, it is sufficient to know which sender owns the lock. This results in the following representation of threads.

```
Thread: TYPE = [#
  status: ThreadStatus,
  partner, lock: ThreadPointer,
  senders_waiting: list[NonNullTP]
#]
```

Though the status flags can be set/cleared individually, e.g., `t`status` returns the status of the thread `t`, one usually considers a certain combination of flags to check whether a thread is in a specific state. For instance, to determine whether two threads are engaged in IPC, the following tests are necessary:

```
inIpc(snd, rcv: NonNullTP)(s : System): bool =
  LET rcv = s`threads(rcv), rcv_stat = rcv`status IN
     rcv_stat`transfer_in_progress ∧ rcv_stat`ipc_in_progress ∧
     ¬rcv_stat`cancel ∧ rcv`partner = snd
```

PVS-functions are explicitly parametrised with a `System` object representing the global state of the machine. Moreover, each function will produce the modified global state as a result. This state is defined as follows:

```
System: TYPE = [#
  current: NonNullTP, threads: [NonNullTP → Thread],
```

```
  error, timeout, fail: bool, seed: nat
#]
```

The field `current` is a pointer to the active thread, `threads` is a 'dereference' function yielding the threads of the system, `error` and `timeout` indicate if an error or a timeout occurred, respectively, and `fail` is set if one of the assertions failed. The field `seed` is explained below.

The manipulation of state information makes specifications needlessly complex. However, with a suitable set of helper functions, one can easily avoid an explicit state object. Particularly, the following composition operation appears to be convenient in our description.

```
SystemFun: TYPE = [System → System]

≫(s1, s2: SystemFun): SystemFun =
  λ (s: System): LET s1s = s1(s) IN
    IF s1s`error THEN s1s ELSE s2(s1s) ENDIF
```

This operation resembles standard function composition. Observe that the second function will not be applied if the first one resulted in an error. Our specification is not monadic, but if was, the composition ≫ operator would become the composition operator ≫ of monads.

In the first step of our approach the set SE of possible side effects is identified. This was done by careful examination of the possible effects concurrent threads can have on a each other, resulting in the following set of preemption actions:

```
PreemptionAction : TYPE =
{ kill,                    % The partner is killed
  timeout,                 % A timeout occurs
  ipc_cancelled,           % IPC has been canceled
  receiver_ready }         % The receiver becomes ready
```

Next, all preemption points are replaced by non-deterministically chosen lists of preemption actions that are executed. Since PVS does not directly support non-determinism, we introduce the following auxiliary function:

```
generatePAs(n: nat): list[PreemptionAction]
```

This function is not further specified. In a proof, this means that we cannot assume anything about the actions appearing in the result list, hence it has to be considered as arbitrary. The argument `n` is necessary for technical reasons: by using different argument values each time `generatePAs` is called, different result lists will be produced. For, had we omitted this argument, `generatePAs` would have been treated as a constant, yielding the same unspecified list of preemption actions everywhere it is called. This explains the existence of the `seed` field in the system state. At each preemption point, the seed is passed to `generatePAs`, and it is incremented.

The effect of preemption actions on the system state is specified by the function `doPreemptionAction`:

```
doPreemptionAction(partner: NonNullTP, allow_timeout: bool)
     (act: PreemptionAction, s:System): System =
  CASES act OF
    ipc_cancelled:  sysThreadExRegs(s`current)(s),
    timeout:        IF allow_timeout
                    THEN timeOut(s`current)(s)
                    ELSE s ENDIF,
    kill:           kill(partner)(s),
    receiver_ready: IF s`current ≠ partner
                    THEN receiverReady(s`current, partner)(s)
                    ELSE s ENDIF,
  ENDCASES
```

The functions `sysThreadExRegs` and `kill` basically set the `cancel` and `dead` flags of the thread status vector, respectively, while `timeOut` sets the `timeout` flag of the system state. The function `receiverReady` sets the bits `ready` and `transfer_in_progress` on the sender and unsets `ready` on the receiver. Ensuring that the sender and the receiver are not the same whenever `receiverReady` is called was necessary to prove certain properties; see Section 5. In Fiasco, this is implicit since it doesn't make sense for a thread to engage in IPC with itself[5].

Finally, we define `preemptionPoint` as the preemption-point function that executes a list of preemption actions.

```
preemptionPoint(partner: NonNullTP, allow_timeout: bool)
     (s:System): System =
   doPAs(partner, allow_timeout)(generatePAs(s`seed))(newSeed(s))

newSeed(s: System): System = s WITH [seed := s`seed + 1]

doPAs(partner: NonNullTP, allow_timeout: bool)
  (pas: list[PreemptionAction])(s: System): System =
     reduce(s, doPreemptionAction(partner, allow_timeout))(pas)
```

The function `reduce` is a predefined list function, similar to fold or fold_left in other languages. In essence, `doPAs` composes the effects of the preemption actions occurring in the list.

These and other basic definitions form the `state` theory. The `ipc` theory contains the model of the C++ functions that implement Fiasco's IPC mechanism. The main function of this theory is

```
doIpc(rcv,snd: NonNullTP, has_rcv,has_snd: bool)(s:System):System =
  IF has_snd ∧ has_rcv
  THEN doIpcSend(rcv, TRUE) >> doIpcReceive(snd)(s)
  ELSIF has_snd THEN doIpcSend(rcv, FALSE)(s)
  ELSIF has_rcv THEN doIpcReceive(snd)(s)
  ELSE s ENDIF
```

[5] And if it tries to, it will get deadlocked waiting for itself to become ready.

A few details of `doIpcSend` will be discussed later; the definition of `doIpcReceive` is unimportant for this chapter.

5 Validating some Properties

This section is based on the PVS theories `prop_wakeup`, `prop_locks`, and `prop_assertions` containing our properties of interest.

Property 1: Receiver woken Consider the send part of an IPC call of a thread t_s that transfers data to a partner thread t_r. In Fiasco the sender is the active part, that is, t_r is sleeping during its receive operation. Sleeping here means that the `ready` flag of t_r is false, causing the scheduler to never select t_r. It is therefore essential that, after the send has been finished, the thread t_s wakes up its partner t_r, such that t_r can be scheduled again. This property is formalised as follows:

```
receiver_woken: LEMMA
   ∀ (partner: Non_Null_TP)(s: System):
      LET sSend = doIpcSend(partner)(s) IN
          ¬ sSend‘error ∧ inIpc(sSend‘current, partner)(sSend)
            ⇒ sSend‘threads(partner)‘state‘ready
```

The property states that if after the execution of `doIpcSend` there is no error on the system state and the sender and the receiver are still engaged, then the `ready` bit of the receiver is set. The proof posed no difficulty and it consisted mainly of definition unfoldings and case distinctions.

Property 2: Lock removed Consider again a thread t_s that wants to engage in a send operation with t_r as receiver. Before actually starting the send, t_s obtains the lock of t_r to make sure that it is the only thread sending to t_r. After the send the lock must of course be released again.

```
lock_removed: LEMMA
  ∀ (rcv_ptr: NonNullTP)(s: System):
      ¬ s‘error ∧ ¬ s‘threads(rcv_ptr)‘status‘invalid AND
      rcv_ptr ≠ nil_thread_ptr ⇒
        LET new_state = doIpcSend(rcv_ptr)(s) IN
           new_state‘threads(rcv_ptr)‘lock = null
```

The property has three requirements, namely, the state of the receiver must be valid, the receiver must not be the nil thread and there should be no error flagged on the initial system state. Under these conditions we were able to prove that after the execution of `doIpcSend`, the lock on the receiver is free.

To reduce the complexity of the proof, five lemmas were created. They assert that the lock is released on each of the possible paths taken by `do_ipc_send`. This decomposition was also very helpful in making the proof more resistant to changes in the model.

Property 3: Assertions passed The Fiasco sources contain some assertions that are spread over the model. When an assertion in the kernel is violated, the system simply halts. We included all the assertions that were expressible in our model, but some assertions referred to things we had abstracted from, like the CPU lock, and thus they were omitted. In total nine assertions were checked and it was in one of them where the bug was found.

To find out if any of them could fail during a call to `sysIpc`, we added the field `fail` to the system state and we defined:

```
assert(b: bool)(s: System): System =
  IF b THEN s ELSE s WITH [fail := TRUE] ENDIF
```

Then the property was stated as shown next.

```
assertions_passed: LEMMA
  ∀ (rcv, snd: NonNullTP, has_rcv,has_snd: bool, s: System):
     ¬doIpc(rcv, snd, has_rcv, has_snd)(s)‘fail
```

The function `doIpcSendPart` contained the assertion causing the failure.

```
doIpcSendPart(partner: NonNullTP, b: bool): SystemFun =
    tryHandshakeReceiver(partner) ≫
    λ(s:System): assert( ¬s‘threads(s‘current)‘status‘polling) ≫
    [...]
```

The problem found is related to the `polling` bit, which is set on the sender when it has to wait for the receiver to become ready. Essentially, the sender *polls* the receiver at intervals to see if it has become ready. Once the receiver is ready and the handshake finishes successfully, this bit should be cleared.

When trying to prove that after a (successful) call to `tryHandshakeReceiver`, the `polling` bit is cleared, we found an execution path in the `doSendWait` function (invoked by `tryHandshakeReceiver`) that did not clear it. After careful examination of the model, the author of that code was contacted and it was confirmed that indeed we had found an error.

But this was not the only complication we faced. There was also an assertion that could not be completely verified within our model due to the abstractions made on the sender. Since we did not model the sender as a separate thread, we could not prove that `inIpc` is commutative, that is, if the sender is engaged in IPC with the receiver, then the receiver is engaged with the sender. An axiom was added to overcome this problem.

Proving this property was quite laborious; 78 other lemmas were used directly or indirectly. To prove that all assertions hold we made lemmas proving that they hold after the execution of each function call that was involved in the IPC system call. For example, the following lemma states that no assertion is violated after a preemption point.

```
assertions_hold_after_preemption_point: LEMMA
  ∀ (partner: NonNullTP, allow_timeout: bool, s: System):
    ¬s‘fail ⇒ ¬preemptionPoint(partner, allow_timeout)(s)‘fail
```

These lemmas in turn use other lemmas, many related to locks and to how each function call affects each of the fields of the thread status.

The proofs were not intrinsically hard but cumbersome. The unfolding of some definitions resulted in proof sequents spanning hundreds of lines in the PVS prover. The following simple pattern can be identified in many of the proofs: unfold definitions and give names to intermediate states (to reduce the size of a sequent) as needed, then prove each branch using other lemmas if needed. Thanks to our lightweight approach to model concurrency, the number of branches was amenable to interactive theorem proving. The only proofs that needed induction were the ones concerning the list of actions that occur at a preemption point and the proofs dealing with the list of senders in the receiver.

6 Case Study Evaluation

In this section we share some reflections and lessons learned from our case study. We also comment on possible directions for future work.

Main lessons learned The case study has validated the applicability of the preemption abstraction approach as a lightweight formal proof method for concurrent code.

Using the proof assistant PVS, we modelled `sys_ipc`: the function that handles all inter-process communication focusing on the interaction between senders and receivers. While constructing the model we followed the source code (its structure and names) as much as possible. We focused both on a few key properties and on the assertions that were contained in the code. Furthermore, we abstracted from some important parts of the system, such as scheduling and Long IPC. Therefore, this case study cannot give a full formal proof of the studied system. However, the proofs of the studied properties significantly increased the confidence in the studied code and, when we found the bug, we could easily point out the corresponding place in the source code where the error occurred.

The code that was analysed is about 3000 lines. The PVS model is about 2000 lines and the proof scripts are another 5000 lines long. Developing the proofs took 2 man-months, but checking that the proofs are correct takes just a few minutes on an ordinary modern computer.

We want to emphasise the fact that the bug was found thanks to an assertion in the code. One usually thinks of assertions as just a run-time check mechanism, but they are more than that: they describe the intended behaviour of the code. We used them to generate properties of our model of the system. Had the code not been instrumented with assertions, we would have probably missed the bug.

The soundness of our approach to model concurrency depends of course on the completeness of the list of actions that may occur at preemption points. We determined the possible events that the environment could have on thread at a preemption point by studying the source code. We are fairly confident that our list is exhaustive, however, a fully formal proof would also verify this assertion.

The preemption abstraction was motivated by the occurrence of explicit preemption points in the studied version of the Fiasco microkernel. There, the atomic blocks were realised by disabling *all* interrupts while enabling them shortly in the preemption points.

Applicability to systems without explicit preemption points The applicability of the preemption abstraction does not depend on the presence of explicit atomic blocks and preemption points in the software. On conventional hardware memory access is atomic, even in systems with multiple processors. For the preemption abstraction it is therefore not relevant whether there are possibly several threads running truly in parallel on several CPU's or not. The important point is, that at the level of memory access, all activity in the system is sequentialised. Therefore, one can think of a memory access as an atomic block with preemption points between memory accesses. Note that one assembly instruction breaks into several atomic blocks if it performs several memory accesses.

Under this interpretation, the number of preemption points will truly be tremendous. One clearly has to formulate the abstract model without writing out every invocation of the preemption-point function. This can easily be achieved with a higher-order combinator that inserts the preemption-point function after each memory access. A legitimate question is, whether it is still possible to verify any property in such a model. In general, the situation is admittedly hopeless. However, systems that have been designed to run in a truly parallel environment without the use of locks are far from the general case.

As an example let us consider a predecessor version of Fiasco that was fully preemptable. There, a timer interrupt could occur after each assembly instruction and induce the scheduling of a different thread. This new thread could potentially modify the state of the interrupted thread. This predecessor version of Fiasco was written in the lock-free programming style [HH01]: to modify a kernel data structure, a thread would first make a private copy, modify this private copy and finally write back the new version in an atomic way (for instance by using the compare-and-swap instruction). If the original data structure has been modified in between, it tries the same procedure again. This way, large portions of the code cannot be affected by parallel running threads, because it only operates on data structures that the other threads do not modify. The calls to the preemption-point function in the abstract model of such code can therefore be treated automatically in the verification environment.

Applicability to model-checking The preemption abstraction can also be applied in a model-checking context. Because model checkers have built-in support for parallel systems the sequentiality of the preemption abstraction is not an advantage per se. However, the reduction of the system $\mathcal{S}$ with its arbitrarily many threads to just one thread should make the state space much smaller. Using the preemption abstraction for model checking remains future work.

Future work A logical next step is to extend the model and prove more properties. We would start by adding preemption and interrupt senders as well as long IPCs. It would also be interesting to prove the completeness of the set of preemption actions. This could be done by modelling all system calls and showing that any effect these calls can have on a running thread has already been considered. During the first phase of this work, we would have benefited from having a tool that, once configured, semi-automatically produces an abstract model. How to create a general tool that yields different models depending on the user's needs, is an interesting research topic with much potential.

7 Related Work

This work is based on the master's thesis of Erik Schierboom [Sch07], in which a first version of the model was developed and the error was spotted.

Fiasco, and in particular its IPC subsystem, has been the subject of several case studies in the application of formal methods to real-world software. In her master's thesis, Endrawaty [End05] modelled the same subsystem of an earlier Fiasco version. She used Promela as specification language and the SPIN model checker [Hol04] to perform simulations and to verify some simple properties. Annamalai [Ann05] extended Endrawaty's model by adding timeouts among other things, and proved more properties, some of which were liveness properties. As in this work, only short IPCs were modelled, however instead of having a lightweight approach to concurrency, they run complete threads in parallel in the model checker leading to huge state spaces. Modelling only two threads where each does only 1 IPC, proving a property took about 8 hours, 2 GBs of RAM and 8 GBs of hard disk. Proving properties about several IPCs or more than two threads was unfeasible. None of these studies found any error in the code. The bug that we found was only introduced later, when René Reussner rewrote Fiasco's IPC in his master thesis [Reu05].

Kolanski and Klein worked closely with the L4 development team to obtain a formalisation of the kernel's application programming interface (API) using the B method [KK06]. Concurrency is modelled using B's parallel composition, hence it is not explicit in their abstract model.

One of the authors of this work was involved in both the VFiasco and the Robin projects [HT05, HT03, Tew07, TWV+08, TVW09]. In both projects the verification of operating-system kernels was attempted, for VFiasco it was the Fiasco microkernel, for Robin it was the Nova micro-hypervisor. At the time of the VFiasco project the Fiasco microkernel was fully preemptable. The Nova micro-hypervisor consists of atomic code blocks with preemption points in between. Both projects concentrated on the modelling and the semantics of certain aspects of the execution environment of these kernels. The verification of larger portions of code was not attempted. Therefore no solution on how to deal with parallelism has been developed in these two projects.

The L4.verified project [Kle09, EKE08, CKS08, Tuc09] attempts the verification of the seL4 kernel. While L4.verified has good chances to finish the first

complete verification of a realistic operating-system kernel, we are not aware of any published information about the interruptibility of the seL4 kernel or the treatment of parallelism in the verification.

Coyotos [SDN+04] is a secure, microkernel-based operating system built in a new systems programming language (BitC) with a well-defined, mechanically-specified semantics. Singularity [HLA+05] is a research operating system at Microsoft Research that aims to build a dependable operating system written in a type-safe language like C# and specified in Sing#, a Spec# extension. These projects are far more comprehensive and long term than our case study, thus a simple comparison would not be fair.

The Verisoft project [AHL+08, DDB08, HP08] aims at the complete verification of a computer system from an e-mail client down to the gate level of the processor. For the verification of their ATAPI disk driver the Verisoft project used a model in which processor steps are interleaved with the steps of the ATAPI device. To simplify the reasoning the interleaved steps are reordered into larger non-interleaved chunks as much as possible. In such scenario it would be possible to apply our abstraction technique.

8 Conclusions

This work presented a lightweight approach to model concurrency which avoids the need of setting up an interleaving semantics and allows one to reason in a non-parallel fashion. This technique is best suited for systems where a component can be affected by its environment at specific points and by well identified actions.

This approach was applied in the modelling of the IPC subsystem of Fiasco microkernel. It enabled proving some properties of the model with reasonable effort. Under the assumption that our high-level model is faithful and that the identified list of actions that may occur at a preemption point is exhaustive, we can ensure that the code honours the properties here studied. During this process we spotted a programming error that, due to its concurrent nature, was hard to be found by testing techniques.

Acknowledgements We would like to thank the operating-systems group at TU Dresden for their support, in particular Rene Reussner and Michael Hohmuth for answering many questions about IPC in Fiasco.

Chapter 3

A Formal Connection between Security Automata and JML Annotations

Revised version of *A Formal Connection between Security Automata and JML Annotations.*
In Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE 2009), held as part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2009).
York, UK, March 22-29, 2009.

A Formal Connection between Security Automata and JML Annotations

Marieke Huisman[1] and Alejandro Tamalet[2] *

[1] University of Twente, The Netherlands
[2] University of Nijmegen, The Netherlands

Abstract. Security automata are a convenient way to describe security policies. Their typical use is to monitor the execution of an application, and to interrupt it as soon as the security policy is violated. However, run-time adherence checking is not always convenient. Instead, we aim at developing a technique to verify adherence to a security policy statically. To do this, we consider a security automaton as specification, and we generate JML annotations that inline the monitor – as a specification – into the application. We describe this translation and prove preservation of program behaviour, *i.e.*, if monitoring does not reveal a security violation, the generated annotations are respected by the program.
The correctness proofs are formalised using the PVS theorem prover. This reveals several subtleties to be considered in the definition of the translation algorithm and in the program requirements.

1 Introduction

With the emergence of a new generation of trusted personal devices (mobile phones, PDAs, *etc.*), the demand for techniques to guarantee application security has become even more prominent. A common approach is to monitor executions with a security automaton [Sch00]. Upon entry or exit of a security-critical method, the security automaton updates its internal state. If it reaches an "illegal" state, the application will be stopped and a security violation will be reported. This approach is particularly suited for properties that are expressed as sequences of legal method calls, such as life cycle properties, or constraints that express how often or under which conditions a method can be called. However, such a monitoring approach is not suited for all applications, depending on their nature and use; sometimes statical means to enforce security are necessary.

A commonly advocated approach is to require that the application carries a correctness proof with it, which can be validated before installing the application on the device. In such a proof carrying code scenario [Nec97], the application provider is required to create this proof.

* This work is partially funded by the IST FET programme of the European Commission, under the IST-2005-015905 Mobius project. This research started while the authors where at INRIA Sophia Antipolis.

Security experts typically express security requirements by a collection of security automata or temporal logic formulae. However, many program verification tools use a Hoare logic style for the specifications (i.e., preconditions and postconditions). Therefore, as a first step towards static verification of such security properties, this chapter proposes a translation from security properties expressed as an automaton (or a safety temporal logic formula, which can be translated into an automaton [Wol02]) into program annotations.

The translation in this chapter is defined for JAVA programs. It is defined in several steps. For each step we provide a correctness proof.

1. We translate a *partial automaton* to a *total automaton* that contains a special trap state to model that an error has occurred. Often it is much more intuitive to specify a security property by a partial automaton, (this avoids cluttering up the automaton with transitions that *"go wrong"*), while for many tools and algorithms, a complete automaton is easier to handle. We show that the behaviour of a program monitored with a partial automaton is equivalent to the behaviour of the program monitored with the total automaton.
2. Using an extension of JML [LPC+09], we generate annotations that capture the behaviour of the total automaton. These are special method-level set-annotations that are evaluated upon entry or exit of a method. We show that run-time monitoring of the program only throws a (new) exception to signal an annotation violation if the monitor reaches the trap state, otherwise the annotated program has the same executions as the monitored program.
3. We inline the set-annotations from the method specification to the method body and prove equivalence of the run-time checking behaviour.

All results in the chapter have been established formally using the PVS theorem prover [ORR+96]. The complete formalisation is available via the second author's website at `http://cs.ru.nl/~marko/research/phds/tamalet/`.

To prove correctness, the order in which method specifications are evaluated is important. Further, we had to add an explicit requirement that `finally` blocks could not override annotation violation exceptions thrown inside `try` or `catch` statements (see also [Hui09]). The last complication that we encountered was how to specify conveniently that specification-only constructs and steps taken by the monitor did not have any side effects on the program state. More detailed information about the proofs is given in Section 4.

Throughout this chapter, we use the *limited SMS* example property of Figure 1 (where ϵ denotes a skip) to illustrate the different translations: the method `sendSMS` can be called and terminate successfully at most `N` times in between calls to `reset`. The counter is not increased if `sendSMS` terminates because of an uncaught exception (with label `exc_exit(sendSMS)`), and `reset` should not be called from within `sendSMS`. Assuming that `reset` can only be invoked by the system, this property can be used to ensure that a third party program cannot send more than `N` messages. Even though very basic, this example is representative of a wide range of important resource-related security properties.

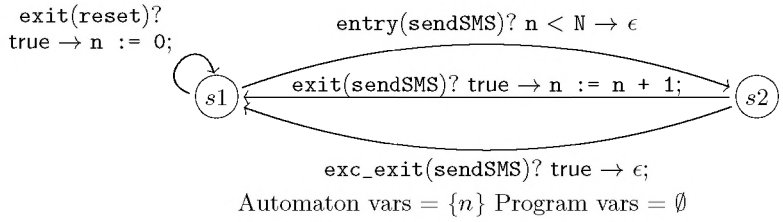


Fig. 1. Example Property Automaton

The rest of this chapter is organised as follows. Section 2 formalises the automaton format and defines completion. Next, Section 3 defines the semantics of monitored and annotated programs. Section 4 defines the translation and proves correctness. Sections 5 and 6 discuss related and future work and conclusions.

2 Modelling Security Properties with Automata

The automata that we use to express security properties are called Property Automata (PA). These are extended finite state machines particularly suited for monitoring, since transitions do not only depend on the automaton's state (i.e., the current control point and a valuation for the automaton's variables), but also on the state of the monitored program. Transitions are labelled with guards, events and a list of actions. Events specify the method whose entry and/or exit is being monitored, with a distinction between normal and exceptional exits. Guards describe the conditions under which a transition can be applied. They depend on

- the automaton state,
- the state of the program that is being monitored, and
- the argument of the method, in case the event is method entry; the result of the method, in case the event is normal method exit; or the exception with which the method returns, in case the event is exceptional method exit.

Actions describe how the automaton state is updated by a transition.

Throughout, we assume that CP and $\mathcal{N}$ are possibly infinite, but countable non-empty sets of control points and names. PA and programs share the definitions of values, types and exceptions, denoted $\mathcal{V}$, $\mathcal{T}$ and $\mathcal{E}$, respectively. These are defined by the following grammar, where $\mathbb{B}$ and $\mathbb{Z}$ denote the standard sets of booleans and integers, respectively[3].

$$\mathcal{V} = \mathsf{B}(b : \mathbb{B}) \mid \mathsf{I}(i : \mathbb{Z}) \mid \mathsf{Null} \mid \mathsf{R}(i : \mathbb{Z}) \mid \mathbb{1} \mid \bot$$
$$\mathcal{T} = \mathsf{Bool} \mid \mathsf{Int} \mid \mathsf{Ref} \mid \mathsf{Void}$$
$$\mathcal{E} = \mathsf{Throwable} \mid \mathsf{RunTimeException} \mid \mathsf{JMLException}$$

[3] We will use a PVS-like notation to declare abstract data types and records (enclosed by `[#` and `#]`). Further, if x is a record with field y, we use x.y to access field y, and x`(#` y $:=$ z `#)` to denote the record x with the field y updated to z.

$$
\begin{aligned}
\mathit{Decl} &= [\#\ \mathsf{type} : \mathcal{T}, \mathsf{name} : \mathcal{N}, \mathsf{init} : \mathcal{V}\ \#] \\
\mathit{Event} &= [\#\ \mathsf{etype} : (\mathsf{entry} \mid \mathsf{exit} \mid \mathsf{exc\_exit}), \mathsf{mname} : \mathcal{N}\ \#] \\
\mathit{Trans} &= [\#\ \mathsf{source}, \mathsf{dest} : \mathit{CP}, \mathsf{event} : \mathit{Event}, \mathsf{action} : ([\#\ \mathsf{target} : \mathcal{N}, \mathsf{expr} : \mathit{Expr}\ \#])^*, \\
&\qquad \mathsf{guard} : \mathit{PAState} \times \mathit{PState} \times (\mathcal{V} \mid \mathcal{E}) \to \mathbb{B}\ \#] \\
\mathit{PA} &= [\#\ \mathsf{name}, \mathsf{clname} : \mathcal{N}, \mathsf{cps} : \mathcal{P}(\mathit{CP}), \mathsf{init} : \mathit{CP}, \mathsf{events} : \mathcal{P}(\mathit{Event}), \\
&\qquad \mathsf{pa\_var\_decl} : \mathcal{P}(\mathit{Decl}), \mathsf{prog\_var\_decl} : \mathcal{P}(\mathit{Decl}), \mathsf{trans} : \mathcal{P}(\mathit{Trans})\ \#]
\end{aligned}
$$

Fig. 2. Formal Definition of PA

The type Void, inhabited by $\mathbb{1}$, models methods without results; a reference can be Null or contain a number representing the location where the object is stored; $\perp$ is used to denote the outcome of an expression whose evaluation is undefined (in JAVA this would typically result in an exception).

A PA consists of

- a name,
- a class name, to specify which class is being monitored,
- a finite set of control points,
- an initial control point,
- a set of events, to specify which methods are being monitored,
- a set of PA variable declarations, to describe the internal state of the automaton,
- a set of program variable declarations, to specify which program variables will be inspected by the monitor, and
- a set of transitions.

Transitions relate source and target control points; they are labelled with events, a guard and a list of actions. An event is a tuple of an event type (entry, exit or exceptional exit), and a method name. Each action assigns the result of an expression (containing both program and PA variables) to a PA variable. Notice that we only monitor classes here. This is often the case in practice, because security-critical methods are often static API methods. However, a more precise formalisation of JAVA's semantics would allow to monitor objects as well. Figure 2 shows the main components of the formal PA definition.

We require a PA to be *deterministic*, i.e., for every source control point and event there is always at most one guard that holds. Notice that it is not obvious how to transform a non-deterministic PA into a deterministic one, because the actions made by the overlapping transitions might differ. A PA is *total* if for any source control point and event, there is always a guard that holds; otherwise it is *partial*. Every deterministic PA can be completed into a total one (by function complete): add a special control point halted, together with transitions for every control point and every event to halted, where the guard is the negation of the disjunction of all other guards for this control point and event. Additionally, add unconditional transitions from halted to halted for every possible event.

A PA is *wellformed* if:

- variable names are unique and are not reserved words,
- guards do not have side effects,
- guards and actions only use declared variables, and
- control points and events in transitions are declared.

The state of a PA consists of a current control point, and the store of automaton variables: $\mathit{PAState} = [\#\,\mathsf{current} : \mathit{CP}, \mathsf{store_A} : \mathit{Store}\,\#]$. Note that the program store is not part of the automaton state, however, as we will see next, the program state is used by the transition function. Given PA a, the transition function Δ_a specifies how an automaton state σ_A is updated for a given program state σ_P, an event e, and a value or exception v (where ε is the arbitrary choice operator, and apply is a function that updates the automaton store according to a list of actions in the obvious way).

$$
\begin{array}{l}
\Delta_a : \mathit{PAState} \times \mathit{PState} \times \mathit{Event} \times (\mathcal{V} \mid \mathcal{E}) \hookrightarrow \mathit{PAState} \\
\Delta_a(\sigma_A, \sigma_P, e, v) = \\
\quad \mathsf{let}\ t = \varepsilon(\{t \in \mathsf{trans}(a) \mid t.\mathsf{source} = \sigma_A.\mathsf{current} \wedge t.\mathsf{event} = e \wedge \\
\qquad\qquad\qquad\qquad t.\mathsf{guard}(\sigma_A.\mathsf{store_A}, \sigma_P.\mathsf{fields.store}, v)\})\ \mathsf{in} \\
\qquad (\#\ \mathsf{current} := t.\mathsf{dest}, \mathsf{store_A} := \mathsf{apply}(t.\mathsf{action}, \sigma_A.\mathsf{store_A}, \sigma_P.\mathsf{fields.store})\ \#)
\end{array}
$$

In a total PA a, the transition function Δ_a is total. A partial automaton gets stuck on a certain input if and only if the completed PA reaches the state halted.

$$\Delta_a(\sigma_A, \sigma_P, e, v) = \bot \Leftrightarrow \Delta_{\mathsf{complete}(a)}(\sigma_A, \sigma_P, e, v).\mathsf{current} = \mathsf{halted} \qquad (1)$$

Example The property specified in Figure 1 is encoded by the following PA[4].

$$
\begin{array}{l}
(\#\ \mathsf{name} := \mathrm{LimitSMS}, \mathsf{clname} := \texttt{Messaging}, \mathsf{cps} := \{s_1, s_2\}, \mathsf{init} := s_1, \\
\quad \mathsf{events} := \{(\#\ \mathsf{etype} := e, \mathsf{mname} := \texttt{sendSMS}\ \#) \mid e \in \{\mathsf{entry}, \mathsf{exit}, \mathsf{exc\_exit}\}\} \cup \\
\qquad\qquad \{(\#\ \mathsf{etype} := \mathsf{exit}, \mathsf{mname} := \texttt{reset}\ \#)\}, \\
\quad \mathsf{pa\_var\_decl} := \{(\#\ \mathsf{name} := n, \mathsf{type} := \mathsf{Int}, \mathsf{init} := 0\ \#)\}, \mathsf{prog\_var\_decl} := \emptyset, \\
\quad \mathsf{trans} := \{\ (\#\ \mathsf{source} := s_1, \mathsf{dest} := s_2, \mathsf{guard} := \lambda(\sigma_A, \sigma_P, v).\sigma_A(n) < N, \\
\qquad\qquad \mathsf{event} := (\#\ \mathsf{etype} := \mathsf{entry}, \mathsf{mname} := \texttt{sendSMS}\ \#)\ \#), \\
\qquad\quad (\#\ \mathsf{source} := s_2, \mathsf{dest} := s_1, \mathsf{action} := [(\#\ \mathsf{target} := n, \mathsf{expr} := n + 1\ \#)] \\
\qquad\qquad \mathsf{event} := (\#\ \mathsf{etype} := \mathsf{exit}, \mathsf{mname} := \texttt{sendSMS}\ \#)\ \#), \\
\qquad\quad (\#\ \mathsf{source} := s_2, \mathsf{dest} := s_1, \\
\qquad\qquad \mathsf{event} := (\#\ \mathsf{etype} := \mathsf{exc\_exit}, \mathsf{mname} := \texttt{sendSMS}\ \#)\ \#), \\
\qquad\quad (\#\ \mathsf{source} := s_1, \mathsf{dest} := s_1, \mathsf{action} := [(\#\ \mathsf{target} := n, \mathsf{expr} := 0\ \#)], \\
\qquad\qquad \mathsf{event} := (\#\ \mathsf{etype} := \mathsf{exit}, \mathsf{mname} := \texttt{reset}\ \#)\ \#)\}\ \#)
\end{array}
$$

Figure 3 shows the completed PA, where new transitions are dotted.

3 Programs and Semantics

This section first defines an abstract syntax of programs, followed by their semantics. Both are fairly standard, except that the semantics is parametrised on

[4] Where we leave the default guard $\lambda(\sigma_A, \sigma_P, v).\mathsf{true}$ and empty action ϵ implicit.

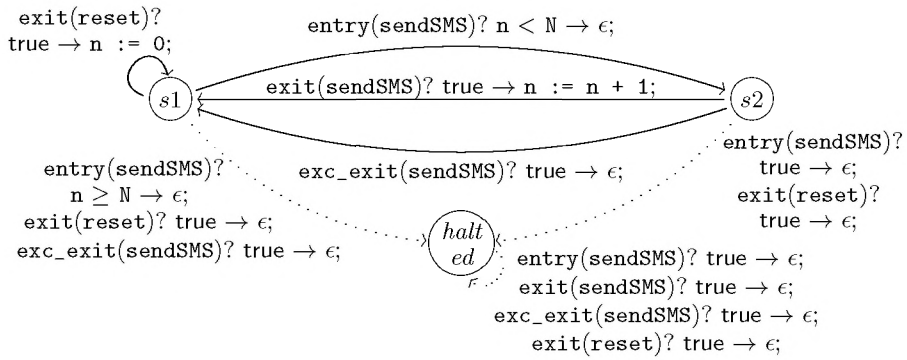


Fig. 3. Automaton of Figure 1, after completion

$$
\begin{aligned}
\mathit{Expr} = {} & \mathsf{Plus}(n_1, n_2 : \mathit{Expr}) \mid \mathsf{Var}_{\mathbb{I}}(n : \mathcal{N}) \mid \mathsf{Not}(b : \mathit{Expr}) \mid \mathsf{And}(b_1, b_2 : \mathit{Expr}) \mid \\
& \mathsf{Eq}(e_1, e_2 : \mathit{Expr}) \mid \mathsf{Var}_{\mathbb{B}}(n : \mathcal{N}) \mid \mathsf{Var}_{\mathbb{R}}(n : \mathcal{N}) \mid \mathsf{CondExpr}(c, e_1, e_2 : \mathit{Expr}) \mid \\
& \mathsf{Assign}(n : \mathcal{N}, e : \mathit{Expr}) \mid \mathsf{Call}(o : \mathit{Expr}, mn : \mathcal{N}, p : \mathit{Expr}) \mid \mathsf{Const}(v : \mathcal{V}) \\
\mathit{Stmt} = {} & \mathsf{Skip} \mid \mathsf{Sequence}(s_1, s_2 : \mathit{Stmt}) \mid \mathsf{IfThenElse}(c : \mathit{Expr}, s_1, s_2 : \mathit{Stmt}) \mid \\
& \mathsf{While}(c : \mathit{Expr}, s : \mathit{Stmt}) \mid \mathsf{StmtExpr}(e : \mathit{Expr}) \mid \mathsf{Throw}(e : \mathcal{E}) \mid \\
& \mathsf{TryCatchFinally}(t : \mathit{Stmt}, e : \mathcal{E}, c, f : \mathit{Stmt}) \mid \mathsf{Set}(n : \mathcal{N}, e : \mathit{Expr}) \mid \\
& \mathsf{CaseSet}(b : \mathsf{list}[\mathit{Expr} \times \mathit{Stmt}]) \mid \mathsf{Assert}(e : \mathit{Expr})
\end{aligned}
$$

Fig. 4. Abstract syntax of expressions and statements

the treatment of specifications. In particular, we define a run-time checking and a monitoring semantics, that evaluate differently upon method call and exit.

3.1 Program Syntax

Our language is a restricted subset of (sequential) JAVA, abstracting away from typical object-oriented features, and in particular from method resolution; instead we assume that the annotated class contains method bodies for the relevant methods, thus method lookup is trivial. We consider only a few exceptions, and assume that methods have only one parameter. We believe, however, that our formalisation contains all constructs that are relevant for proving correctness of our inlining algorithm for class-based monitoring, and implementing the algorithm for the full language is mainly an engineering issue.

Figure 4 defines expressions and statements as a mutually recursive data type (we use the term *body* to denote either an expression or a statement), *e.g.*, Call represents a call to method mn on target o with argument p. Notice that we define several special language constructs to represent JML annotations: Set, to update ghost variables (i.e., specification-only variables), CaseSet, to abbreviate a list of conditional ghost variable updates, and Assert, to evaluate a condition

$$
\begin{aligned}
\mathit{Method} = [\#\ & \mathsf{name} : \mathcal{N}, \mathsf{param} : \mathit{Decl}, \mathsf{lvars} : \mathcal{P}(\mathit{Decl}), \mathsf{body} : \mathit{Stmt}, \\
& \mathsf{res} : \mathit{Expr}, \mathsf{res\_type} : \mathcal{T}, \mathsf{pre}, \mathsf{post} : \mathit{Expr} \to \mathit{Expr}, \\
& \mathsf{pre\_set}, \mathsf{post\_set} : \mathit{Expr} \to \mathit{Stmt}, \mathsf{exc\_set} : \mathcal{E} \to \mathit{Stmt}\ \#] \\
\mathit{Class} = [\#\ & \mathsf{name} : \mathcal{N}, \mathsf{super} : \mathcal{N}_\perp, \mathsf{fields} : \mathcal{P}(\mathit{Decl}), \mathsf{methods} : \mathcal{P}(\mathit{Method}), \\
& \mathsf{inv} : \mathit{Expr}, \mathsf{ghost\_vars} : \mathcal{P}(\mathit{Decl})\ \#] \\
\mathit{Program} = [\#\ & \mathsf{classes} : \mathcal{P}(\mathit{Class})\ \#]
\end{aligned}
$$

Fig. 5. Abstract Syntax for Programs

on the program state. A standard program semantics ignores these statements, whereas the annotated program semantics evaluates them.

CaseSet is not standard; we have introduced it to simplify the translation of the PA's transitions. It works as a case statement where all the variables involved are ghost variables. This constructs can be latter translated into standard JML using the ternary operator ? : for the conditions and Set assignments for the statements.

Figure 5 describes the syntax for methods, classes and programs. To ensure that every method has an appropriate return expression, it is part of the method signature. Furthermore, methods can be annotated with pre- and postconditions, and classes with invariants. To support our annotation generation algorithm, we define special annotations called pre_set, post_set and exc_set. These annotations describe the updates to the ghost variables at method entry, exit and exceptional exit, respectively. In the translation, pre_set will be translated to a series of Set statements at the beginning of the method call and for the post_set and exc_set, the statements go at the end. The motivation for having the annotations at a method-level rather than in the body is that it simplifies their evaluation. Precondition, postcondition and the different method specification-level set annotations have a function type to allow the use of the method parameter, the method result, or the returned exception, respectively.

A program is said to be *wellformed* if

- names of fields, local variables and ghost variables are disjoint and are not reserved words,
- class names are unique,
- method names are unique,
- every variable name that is used is declared, and
- only ghost variables are the target of Set statements.

(We have stated only the wellformedness conditions necessary for our correctness proofs.)

3.2 Natural Semantics

The behaviour of a program is described via a big step semantics. We closely follow Von Oheimb's formalisation of JAVA [vO01], with simplifications wherever

$$\frac{\begin{array}{c}\sigma_0.\mathsf{prog\_state.exc} = \bot \qquad P \vdash \langle o, \sigma_0\rangle \triangleright \langle r, \sigma_1\rangle \qquad P \vdash \langle p, \sigma 1\rangle \triangleright \langle act, \sigma_2\rangle \\ r \neq \mathsf{Null} \qquad md = \mathsf{lookup\_mthd}(P, r, mn) \\ old\_lvs = \sigma_2.\mathsf{prog\_state.store.loc\_vars} \quad \sigma_3 = \mathsf{update\_lvs}(\sigma_2, r, md.\mathsf{lvars}, md.\mathsf{param}, act) \\ \gamma_{\text{IN}}(P, md, r, \mathsf{Const}(act), \sigma_3, \sigma_4) \qquad P \vdash \langle md.body, \sigma_4\rangle \triangleright \langle \mathbb{1}, \sigma_5\rangle \\ P \vdash \langle md.\mathsf{res}, \sigma_5\rangle \triangleright \langle v, \sigma_6\rangle \qquad \gamma_{\text{NORM}}(P, md, r, \mathsf{Const}(v), \sigma_6, \sigma_7)\end{array}}{P \vdash \langle \mathsf{Call}(o, mn, p), \sigma_0\rangle \triangleright \langle v, \sigma_7(\mathsf{prog\_state.store.loc\_vars} := old\_lvs)\rangle}$$

Fig. 6. Evaluation rule for normal termination of method calls

possible, due to our simplified program syntax. A judgement $P \vdash \langle e, \sigma\rangle \triangleright \langle v, \sigma'\rangle$ means that the body e evaluates to v, while transforming the state σ into σ', in the context of the program P. Note that v is $\mathbb{1}$ for normally terminating *statements*, while v is $\bot$ whenever evaluation finishes in an exceptional state.

A basic program state *PState* is composed of an optional exception and a store. The store maps every field and local variable to a value.

$$\begin{array}{l} \mathit{PState} = \texttt{[\#}\ \mathsf{exc} : \mathit{Excp}_{\bot}, \mathsf{store} : \mathit{PStore}\ \texttt{\#]} \\ \mathit{PStore} = \texttt{[\#}\ \mathsf{fields} : \mathcal{N} \mapsto \mathcal{V}, \mathsf{loc\_vars} : \mathcal{N} \mapsto \mathcal{V}\ \texttt{\#]} \end{array}$$

The evaluation rules for annotated and monitored programs are the same for most constructs; they differ in places like the evaluation of JML annotations or the update of the PA state. Instead of defining two similar semantics we have defined one where we abstract the differences as parameters. Two of the parameters are *FullProgram* and *FullState*, which can be thought as the base classes for programs and states, respectively. For each instantiation we give mappings program and prog_state to the basic program type *Program* and the basic program state *PState*. Further, we add parameters that specify the actions that are taken upon method entry or (normal or exceptional) exit (γ_{IN}, γ_{NORM}, and γ_{EXC}, respectively), and the handling of annotations (δ_{SET}, δ_{ASSERT}, and δ_{CASE}). In a standard program semantics, where specifications are ignored, these are all instantiated with the identity relation.

The evaluation rules are fairly standard, and we refer to Von Oheimb and the PVS formalisation for more details. Evaluation of normally terminating method calls is described by the following rule (where for clarity of presentation, we left out several checks that intermediate states are not exceptional).

First the receiver is evaluated, resulting in non-null reference r. Next, the parameter is evaluated, resulting in value *act*. Using r, the method definition *md* is looked up. The local variable store is updated assigning r to `this`, initialising the method's local variables and assigning the actual parameter to the formal parameter. The old local variable store is remembered as *old_lvs*. Next, an appropriate action upon method entry is taken, as specified by the relation γ_{IN}. Then the method body, and method result expression are evaluated. Since this rule applies to normal method termination only, the parameter for normal

method termination γ_{NORM} is evaluated. Last, the local store is set back to *old_lvs*. In addition, rules exist that specify behaviour of a method call when it is called upon a null reference, the body contains an uncaught exception *etc.*

Annotated program semantics

The program state of an annotated program is extended with a store for ghost variables:

$$AState = [\#\ \mathsf{pstate} : PState, \mathsf{ghost\_vars} : \mathcal{N} \mapsto \mathcal{V}\ \#]$$

The types *FullProgram* and *Program* coincide, while *FullState* is instantiated as *AState*, and the mapping prog_state is defined as pstate. Figure 7 shows the instantiations of the semantics parameters (presented in rule format). The relation γ_{IN} uses the auxiliary relation β which checks a boolean expression e and raises a special JMLException if it evaluates to false. Upon method entry, the class invariant and precondition are evaluated. We assume that lookup_inv returns the complete class invariant, including those invariants that are inherited from superclasses. If they fail, a JMLException is thrown, otherwise the method's pre_set statement is executed. Finally, we ensure that the program store is not changed. The relations γ_{NORM} and γ_{EXC} are similar. Note that γ_{EXC} does not check the postcondition because an exception has occurred and that if the evaluation of exc_set or the invariant raise a new exception, it is overridden by the original exception. The function δ_{SET} updates a ghost variable: it first evaluates the expression and if this did not result in an exceptional state, it updates the value of the ghost variable[5] appropriately.

Monitored program semantics

The parametrised program semantics is also instantiated for monitored programs. This semantics is only defined when the PA is compatible with the program. PA a is said to be compatible with a program P, denoted $a \sqsubseteq P$, if

- the program contains the class c that is being monitored,
- all variables declared as program variables in a are fields of the class c with the correct type, and
- every event name corresponds to a method in the class.

A monitored program is a product of a PA and a program. The state of a monitored program consists of the states of the PA and the program (including ghost variables)[6], and a flag stuck. If the PA is partial, the flag stuck is set

[5] Where $\tau(\mathsf{ghost\_vars}.n := v)$ abbreviates that the value of $\mathsf{ghost\_vars}(n)$ in τ is updated to v.

[6] For convenience, we assume that a monitored program also evaluates annotations, but this instantiation is in fact orthogonal to the annotated program semantics.

$$\frac{act \neq \bot \quad inv = \mathsf{lookup\_inv}(P, r) \quad \beta(P, inv, \sigma_1, \tau_1) \quad \beta(P, md.\mathsf{pre}(act), \tau_1, \tau_2) \quad P \vdash \langle md.\mathsf{pre\_set}(act), \tau_1 \rangle \triangleright \langle v, \tau_2 \rangle \quad v \in \{\bot, \mathbb{1}\} \quad \sigma_1.\mathsf{pstate.store} = \sigma_2.\mathsf{pstate.store}}{\gamma_{\text{IN}}(P, md, r, act, \sigma_1, \sigma_2)}$$

$$\frac{act \neq \bot \quad P \vdash \langle md.\mathsf{post\_set}(act), \sigma_1 \rangle \triangleright \langle v, \tau_1 \rangle \quad v \in \{\bot, \mathbb{1}\} \quad inv = \mathsf{lookup\_inv}(P, r) \quad \beta(P, inv, \tau_1, \tau_2) \quad \beta(P, md.\mathsf{post}(act), \tau_2, \sigma_2) \quad \sigma_1.\mathsf{pstate.store} = \sigma_2.\mathsf{pstate.store}}{\gamma_{\text{NORM}}(P, md, r, act, \sigma_1, \sigma_2)}$$

$$\frac{\begin{array}{c} act = \bot \quad P \vdash \langle md.\mathsf{exc\_set}(act), \sigma_1(\mathsf{exc} := \bot) \rangle \triangleright \langle v, \tau_1 \rangle \quad v \in \{\bot, \mathbb{1}\} \\ inv = \mathsf{lookup\_inv}(P, r) \quad \beta(P, inv, \tau_1, \tau_2) \\ \text{if } \tau_2.\mathsf{pstate.exc} = \bot \text{ then } \sigma_2 = \tau(\mathsf{exc} := \sigma_1.\mathsf{pstate.exc}) \text{ else } \sigma_2 = \tau_2 \\ \sigma_1.\mathsf{pstate.store} = \sigma_2.\mathsf{pstate.store} \end{array}}{\gamma_{\text{EXC}}(P, md, r, act, \sigma_1, \sigma_2)}$$

$$\frac{P \vdash \langle e, \sigma_1 \rangle \triangleright \langle v, \tau \rangle \quad \text{if } v = \mathsf{B(true)} \text{ then } \sigma_2 = \tau \text{ else } \sigma_2 = \tau(\mathsf{exc} := \mathsf{JMLException})}{\beta(P, e, \sigma_1, \sigma_2)}$$

$$\frac{P \vdash \langle e, \sigma_1 \rangle \triangleright \langle v, \tau \rangle \quad \text{if } \tau.\mathsf{pstate.exc} = \bot \text{ then } \sigma_2 = \tau(\mathsf{ghost\_vars}.n := v) \text{ else } \sigma_2 = \tau}{\delta_{\text{SET}}(P, \mathsf{Set}(e, n), \sigma_1, \sigma_2)}$$

Fig. 7. Instantiation of semantics for run-time annotation evaluation

when Δ_a is not defined for a certain input. If the flag is set, this means that the security policy is violated, and the program should be stopped (by some external observer). If the PA is total, the stuck flag will never be set. Instead, violation of the security policy is modelled by the PA reaching the trap state halted (in which case the external observer again is supposed to stop execution).

$$\begin{aligned} \mathit{MProgram} &= [\#\ \mathsf{pa} : \mathit{PA}, \mathsf{program} : \mathit{Program}\ \#] \\ \mathit{MState} &= [\#\ \mathsf{pa\_state} : \mathit{PAState}, \mathsf{pstate} : \mathit{PState}, \mathsf{ghost\_vars} : \mathcal{N} \mapsto \mathcal{V}, \mathsf{stuck} : \mathbb{B}\ \#] \end{aligned}$$

Thus, *FullProgram* gets instantiated as *MProgram* and *FullState* as *MState*, with mappings program and pstate. Now we can give appropriate instantiations for the γ- and δ-relations. The δ-relations are the same as for the annotated program semantics, but the γ-relation also updates the state of the monitor.

$$\frac{\gamma^{\mathsf{AP}}_{\text{IN}}(P, md, r, act, \sigma_1, \tau) \quad \text{if } \tau.\mathsf{pstate.exc} = \bot \text{ then } \sigma_2 = \gamma_{\text{PA}}(\mathsf{entry})(P, md, act, \tau) \text{ else } \sigma_2 = \tau \; c}{\gamma_{\text{IN}}(P, md, r, act, \sigma_1, \sigma_2)}$$

$$\frac{\begin{array}{c} act \neq \bot \quad P \vdash \langle md.\mathsf{post\_set}(act), \sigma_1 \rangle \triangleright \langle v, \tau_1 \rangle \quad v \in \{\bot, \mathbb{1}\} \\ \tau_2 = \gamma_{\text{PA}}(\mathsf{exit})(P, md, act, \tau_1) \quad inv = \mathsf{lookup\_inv}(P, r) \quad \beta(P, inv, \tau_2, \tau_3) \\ \beta(P, md.\mathsf{post}(act), \tau_3, \sigma_2) \quad \sigma_1.\mathsf{pstate.store} = \sigma_2.\mathsf{pstate.store} \end{array} c}{\gamma_{\text{NORM}}(P, md, r, act, \sigma_1, \sigma_2)}$$

$$\frac{\begin{array}{c} act = \bot \quad P \vdash \langle md.\mathsf{exc\_set}(act), \sigma_1(\mathsf{exc} := \bot) \rangle \triangleright \langle v, \tau_1 \rangle \quad v \in \{\bot, \mathbb{1}\} \\ \tau_2 = \gamma_{\text{PA}}(\mathsf{exc\_exit})(P, md, act, \tau_1) \quad inv = \mathsf{lookup\_inv}(P, r) \quad \beta(P, inv, \tau_2, \tau_3) \\ \text{if } \tau_3.\mathsf{pstate.exc} = \bot \text{ then } \sigma_2 = \tau(\mathsf{exc} := \sigma_1.\mathsf{pstate.exc}) \text{ else } \sigma_2 = \tau_2 \\ \sigma_1.\mathsf{pstate.store} = \sigma_2.\mathsf{pstate.store} \end{array}}{\gamma_{\text{EXC}}(P, md, r, act, \sigma_1, \sigma_2)}$$

where $\gamma_{\text{IN}}^{\mathsf{AP}}$ is the instantiation of γ_{IN} for annotated programs, as defined in Figure 7, and

$$\begin{aligned}\gamma_{\text{PA}}(ev)(P, md, act, \sigma) = \mathsf{let}\ \ & e = (\#\,\mathsf{etype} := ev, \mathsf{mname} := md.\mathsf{name}\,\#), \\ & \tau = \Delta_{P.\mathsf{pa}}(\sigma.\mathsf{pa\_state}, \sigma.\mathsf{prog\_state}, e, act)\ \mathsf{in} \\ & \mathsf{if}\ \sigma.\mathsf{stuck} \vee \tau = \bot\ \mathsf{then}\ \sigma(\mathsf{stuck} := \mathsf{true})\ \mathsf{else}\ \sigma(\mathsf{pa\_state} := \tau)\end{aligned}$$

4 Annotation Generation

Given a security property encoded as a PA, the annotation generation procedure generates JML-annotations that capture this property, i.e., if the program respects the property encoded by the monitor, then it does not violate the generated annotations. As explained above, the procedure is defined in several steps:

1. the monitor is completed,
2. the annotations are generated at the method specification level, as special set-annotations, and
3. the method specification-level set-annotations are inlined in the method body.

Notice that the special CaseSet annotation could be translated into standard JML annotations as well.

For each step we prove that the new program simulates the old program, i.e., we show for every translation step α there exists a relation R such that:

$$\begin{aligned}\forall b, \sigma_1, \sigma_2, \tau_1, v_1. P \vdash \langle b, \sigma_1\rangle \triangleright \langle v_1, \sigma_2\rangle \wedge R(\sigma_1, \tau_1) \Rightarrow \\ \exists \tau_2, v_2. \alpha(P) \vdash \langle b, \tau_1\rangle \triangleright \langle v_2, \tau_2\rangle \wedge R(\sigma_2, \tau_2)\end{aligned}$$

Additionally, we show that the initial program states are related by R, and from this we can conclude that for any reachable state of the monitored program, there exists a related state, reachable in the translated program. As a side remark, for translation steps (i) and (iii), we can even prove that relation R is a bisimulation, while for step (ii) we can only prove existence of a simulation (since non-terminating monitored programs – for which no derivation exists in the natural semantics – might terminate after annotation generation, because of an annotation violation).

A natural way to prove the simulation is by induction over the derivation length. However, induction can only be applied when the body is unchanged. Since the translation introduces new (ghost) variables to encode the PA, this is not always the case. For these cases, separate preservation lemmas have to be proven. Further, to be able to complete the proof, we need to ensure that in both bodies the same branches of conditional expressions and statements are taken, and that the same values get assigned to the store. Therefore, we prove a stronger result, adding that also the values v_1 and v_2 are the same (for step (ii): provided the monitor did not reach the halted state).

Completion of the automaton

The first translation step does not change the program itself, it only completes the PA. Suppose that P is a monitored program, where the monitor $P.\mathsf{pa}$ is deterministic and wellformed. Then the translation to a monitored program with a total PA, $\alpha_1(P)$, is defined as:

$$\alpha_1(P) = (\#\ \mathsf{pa} := \mathsf{complete}(P.\mathsf{pa}), \mathsf{program} := P.\mathsf{program}\ \#)$$

The relation that is preserved between executions of P and $\alpha_1(P)$ is the following (where σ is a state of P and τ is a state of $\alpha_1(P)$):

$$\begin{aligned} R(\sigma,\tau) = {} & (\mathsf{if}\ \sigma.\mathsf{stuck}\ \mathsf{then}\ \tau.\mathsf{pa\_state.current} = \mathsf{halted} \\ & \quad \mathsf{else}\ \sigma.\mathsf{pa\_state.current} = \tau.\mathsf{pa\_state.current}) \wedge \neg\tau.\mathsf{stuck} \wedge \\ & (\sigma.\mathsf{pa\_state.store_A} = \tau.\mathsf{pa\_state.store_A}) \wedge \\ & (\sigma.\mathsf{pstate} = \tau.\mathsf{pstate}) \wedge (\sigma.\mathsf{ghost\_vars} = \tau.\mathsf{ghost\_vars}) \end{aligned}$$

To prove that this relation is preserved for any body b, we use equivalence (1) on Page 42 and we observe further that

- if stuck has been set, it remains set,
- for a total PA, if halted is reached, it is never left, and
- for a total PA, stuck is never set.

Formally, where P is a monitored program, and Q is a monitored program with total PA:

$$\begin{aligned} \sigma_1.\mathsf{stuck} \wedge P \vdash \langle b, \sigma_1\rangle \triangleright \langle v, \sigma_2\rangle &\Rightarrow \sigma_2.\mathsf{stuck} \\ \sigma_1.\mathsf{pa\_state.current} = \mathsf{halted} \wedge Q \vdash \langle b, \sigma_1\rangle \triangleright \langle v, \sigma_2\rangle &\Rightarrow \sigma_2.\mathsf{pa\_state.current} = \mathsf{halted} \\ \neg\sigma_1.\mathsf{stuck} \wedge Q \vdash \langle b, \sigma_1\rangle \triangleright \langle v, \sigma_2\rangle &\Rightarrow \neg\sigma_2.\mathsf{stuck} \end{aligned}$$

To illustrate how the annotation generation algorithm works on the LimitSMS automaton in Figure 1, assume we have declared a class `Messaging`, containing the methods used by the automaton, plus a method `receiveSMS`. Applying translation α_1 means that this class, instead of being monitored by the partial PA in Figure 1, is monitored by the total PA in Figure 3.

From PA to annotations

Figure 8 contains the formal definition of the second translation step: from PA to method-level set-annotations. Given a monitored program P where $P.\mathsf{pa}$ is total, the annotation generation algorithm α_2 applies $\alpha_{2,C}$ to all classes. This function checks whether the class is the one being monitored. If so, appropriate ghost variables are added to the class using the function new_vars (see the PVS formalisation for its formal definition). Basically

- for each automaton control point q, a (final) ghost variable declaration `q` is generated, initialised to a unique value; (i.e., we assume we have a function unique that maps each control point to a unique integer),

$$
\begin{aligned}
\alpha_2(P) &= (\#\ \mathsf{classes} := \{\alpha_{2,\mathcal{C}}(c, P.\mathsf{pa}) \mid c \in P.\mathsf{program.classes}\}\ \#) \\
\alpha_{2,\mathcal{C}}(c,a) &= \mathsf{if}\ c.\mathsf{name} \neq a.\mathsf{clname}\ \mathsf{then}\ c \\
&\quad \mathsf{else}\ c\ (\#\ \mathsf{ghost\_vars} := c.\mathsf{ghost\_vars} \cup \mathsf{new\_vars}(a) \\
&\qquad \mathsf{inv} := \mathsf{And}(\mathsf{Not}(\mathsf{Eq}(\mathsf{cp}, \mathtt{halted})), c.\mathsf{inv}) \\
&\qquad \mathsf{methods} := \{\alpha_{2,\mathcal{M}}(m,a) \mid m \in c.\mathsf{methods}\}\ \#) \\
\alpha_{2,\mathcal{M}}(m,a) &= m\ (\#\ \mathsf{pre\_set} := m.\mathsf{pre\_set}; \alpha_{2,\mathcal{E}}(\mathsf{entry}, m.\mathsf{name}, a); \\
&\qquad\qquad \mathsf{Assert}(\mathsf{Not}(\mathsf{Eq}(\mathsf{cp}, \mathtt{halted}))), \\
&\qquad \mathsf{post\_set} := m.\mathsf{post\_set}; \alpha_{2,\mathcal{E}}(\mathsf{exit}, m.\mathsf{name}, a) \\
&\qquad \mathsf{exc\_set} := m.\mathsf{exc\_set}; \alpha_{2,\mathcal{E}}(\mathsf{exc\_exit}, m.\mathsf{name}, a)\ \#) \\
\alpha_{2,\mathcal{E}}(e,n,a) &= \alpha_{2,\mathcal{T}}(\{t \mid t \in a.\mathsf{trans} \wedge t.\mathsf{etype} = (\#\ \mathsf{event} := e, \mathsf{mname} := m\ \#)\}, a) \\
\alpha_{2,\mathcal{T}}(ts,a) &= \mathsf{CaseSet}(\{(\mathsf{Eq}(\mathsf{cp}, \mathsf{q}), \alpha_{2,\mathcal{S}}(ts,q)) \mid \mathsf{q} \in a.\mathsf{cps}\}) \\
\alpha_{2,\mathcal{S}}(ts,q) &= \mathsf{CaseSet}(\{(t.\mathsf{guard}, \mathsf{Set}(\mathsf{cp}, t.\mathsf{dest}; t.\mathsf{action})) \mid t \in ts \wedge t.\mathsf{source} = \mathsf{q}\})
\end{aligned}
$$

Fig. 8. Formal definition of translation PA into annotations

- a ghost variable cp is declared, initialised to the value of the ghost variable representing the initial control point, and
- for each automaton variable declaration, a ghost variable is declared with corresponding name, type and initialisation.

Further, $\alpha_{2,\mathcal{C}}$ adds the condition that the current control point should not be halted to the class invariant[7], and it annotates all methods in the class using $\alpha_{2,\mathcal{M}}$. For each method, pre_set, post_set and exc_set are extended with updates to the ghost variables encoding the automaton. In addition, at the end of pre_set, an Assert statement is added to verify whether the transition reached the halted state: in that case program execution should terminate immediately. Without this Assert, the property violation would only be detected after the body is executed. To encode the updates to the ghost variables, $\alpha_{2,\mathcal{E}}$ computes the set of relevant transitions (i.e., those where the event and method name correspond). For these transitions, a CaseSet statement is generated, where the different cases correspond to the current control point being equal to a control point q, for any q in the automaton. For each such q, $\alpha_{2,\mathcal{S}}$ selects the transitions where t.source is q and generates a CaseSet statement, that tests whether the guard holds, and if so, sets the control point cp to t.dest, and executes the actions associated with this transition. Notice that the order in which the different cases are generated is not important: since the PA is total and deterministic there is always exactly one case that applies.

The formalisation does not specify how guards and actions are translated. Instead, we assume there exists a translation into expressions in the programming language that

- are wellformed,

[7] For readability, we do not explicitly write the translation from PA control points to ghost variables.

```
class Messaging {
//@ ghost int halted = 0, s1 = 1, s2 = 2, N = 5, cp = s1, n = 0;
//@ public invariant cp != halted;

/*@ pre_set  CaseSet [(cp == s1, CaseSet [(n < N, cp = s2),
                                          (n >= N, cp = halted)]),
                    (cp == s2, CaseSet [(true, cp = halted)]),
                    (cp == halted, CaseSet [(true, cp = halted)])];
            Assert cp != halted;
    post_set CaseSet [(cp == s1, CaseSet [(true, cp = halted)]),
                    (cp == s2, CaseSet [(true, cp = s1; n = n + 1)]),
                    (cp == halted, CaseSet [(true, cp = halted)])];
    exc_set  CaseSet [(cp == s1, CaseSet [(true, cp = halted)]),
                    (cp == s2, CaseSet [(true, cp = s1)]),
                    (cp == halted, CaseSet [(true, cp = halted)])]; @*/
void sendSMS(){/* body sendSMS */}

/*@ pre_set  CaseSet []; Assert cp != halted;
    post_set CaseSet []; exc_set  CaseSet []; @*/
void receiveSMS(){/* body receiveSMS */}

/*@ pre_set  CaseSet []; Assert cp != halted; exc_set CaseSet [];
    post_set CaseSet [(cp == s1, CaseSet [(true, cp = s1; n = 0)]),
                    (cp == s2, CaseSet [(true, cp = halted)]),
                    (cp == halted, CaseSet [(true, cp = halted)])]; @*/
void reset(){/* body reset */} }
```

Fig. 9. Method-level set annotations generated for class Messaging

- give the same result,
- do not have side effects,
- do not throw exceptions, and
- do not contain method calls.

From this we can derive that in the annotated program, the generated statements in pre_set can only throw a JMLException (because of the concluding Assert), while the generated statements in post_set and exc_set do not throw any exception.

To illustrate the translation on our running example consider again the class `Messaging` and the completed PA, encoding the *limited SMS* policy, in Figure 3. Figure 9 shows the generated annotations that result from applying translation $\alpha_{2,C}$ on this class and this PA. Notice that for methods and events that are not involved in the property, an empty CaseSet is generated – this is equivalent to a Skip statement.

To show correctness of the translation, we show that the following relation is preserved (where P is the monitored program, σ a state of the monitored program, and τ a state of the annotated program):

$$
\begin{aligned}
R(\sigma,\tau) = {} & \neg\sigma.\mathsf{stuck} \wedge \\
& \text{if } \sigma.\mathsf{pa\_state.current} = \mathsf{halted} \text{ then } \tau.\mathsf{pstate.exc} = \mathsf{JMLException} \text{ else } S(\sigma,\tau) \\
S(\sigma,\tau) = {} & (\mathsf{unique}(\sigma.\mathsf{pa\_state.current}) = \tau.\mathsf{ghost\_vars}(\mathsf{cp})) \wedge \\
& \forall q \in P.\mathsf{pa.cps}.\ (\mathsf{unique}(q) = \tau.\mathsf{ghost\_vars}(\mathsf{q})) \wedge \\
& \forall n \in \mathcal{N}.(\sigma.\mathsf{pa\_state}(n) \neq \bot \Rightarrow \sigma.\mathsf{pa\_state}(n) = \tau.\mathsf{ghost\_vars}(n)) \wedge \\
& \sigma.\mathsf{pstate} = \tau.\mathsf{pstate} \wedge \\
& \forall n \in \mathcal{N}.(\sigma.\mathsf{ghost\_vars}(n) \neq \bot \Rightarrow \sigma.\mathsf{ghost\_vars}(n) = \tau.\mathsf{ghost\_vars}(n))
\end{aligned}
$$

This relation specifies that if the monitor has reached control point halted, the annotated program must have thrown a JMLException. Otherwise, the state of the annotated program corresponds to the state of the original program, extended with the modelling of the monitor's state. This means that the program states (fields, local variables and exceptions) have to coincide, just as the values of the ghost variables that are declared in the original program P. Further, the current control point is represented by the value stored in ghost variable cp, and all PA control points and variables correspond to ghost variables. Notice that if an annotation already present in P causes a JMLException, both the monitored and the annotated program will throw it. Therefore, we cannot prove that the annotated program throws a JMLException *if and only if* halted is reached.

To prove that this relation is preserved, it is strengthened with the following property: if the control point is not halted, then the derivations produce the same value. The crucial part in the proof is of course what happens upon method call and termination. For example, when a method is called, first the invariant and the precondition are evaluated. Assuming that halted is not yet reached, the new conjunct of the invariant evaluates to true, and induction allows to derive that after evaluation of the precondition, the states are related by R. Next, the original pre_set annotations are evaluated, and again the induction hypothesis allows to conclude that the resulting states are related. Next, the monitored program makes a PA transition, and the annotated program executes the newly generated set annotations, followed by an Assert to check whether halted has been reached. Here we cannot use the induction hypothesis, but instead we show manually that relation R is preserved. Notice that in post_set or exc_set we do not have an Assert statement. Since the invariant is evaluated immediately after the set-annotations, the reaching of halted will be detected immediately. For this part of the proof it is crucial that the newly added invariant is evaluated first.

Finally, to complete the proof, we have to add a restriction to programs. We follow the JAVA Language Specification in describing its behaviour [AGH05]. This means in particular that if the *finally* block in the statement terminates abnormally (because of an exception, or any other reason for abrupt completion), it overrides a possible exception thrown in the *try* or *catch* block. Thus, for example, if halted is reached in the *try* block, and hence a JMLException is thrown, this exception might be overwritten by an exception thrown in the *finally* block (see also [Hui09] for a discussion of this problem), which would mean that the violation of the security policy is not signalled to the user, and instead execu-

$$
\begin{aligned}
\alpha_3(P) &= (\#\ \mathsf{classes} := \{\alpha_{3,\mathcal{C}}(P,c) \mid c \in P.\mathsf{program.classes}\}\ \#) \\
\alpha_{3,\mathcal{C}}(P,c) &= c(\#\ \mathsf{methods} := \{\alpha_{3,\mathcal{M}}(P,m) \mid m \in c.\mathsf{methods}\}\ \#) \\
\alpha_{3,\mathcal{M}}(P,m) &= m(\#\ \mathsf{pre\_set} := \mathsf{Skip}, \mathsf{post\_set} := \mathsf{Skip}, \mathsf{exc\_set} := \mathsf{Skip}, \\
&\qquad \mathsf{lvars} := \{\mathsf{result}\} \cup m.\mathsf{lvars}, \mathsf{res} := \mathsf{lookup}(\mathsf{result}), \\
&\qquad \mathsf{body} := \mathsf{TryCatchFinally}(\\
&\qquad\qquad \mathsf{TryCatchFinally}(m.\mathsf{pre\_set}; m.\mathsf{body}; \\
&\qquad\qquad\qquad \mathsf{Assign}(\mathsf{result}, m.\mathsf{res}); m.\mathsf{post\_set} \\
&\qquad\qquad \mathsf{Throwable}, m.\mathsf{exc\_set}, \mathsf{Skip}), \\
&\qquad\quad \mathsf{RunTimeException}, m.\mathsf{exc\_set}, \mathsf{Skip})\ \#)
\end{aligned}
$$

Fig. 10. Formal definition of annotation inlining for methods

tion continues (with another exception). To avoid this, for all TryCatchFinally statements in the program, we require that if the *try* or *catch* block throws a JMLException, the whole statement also terminates exceptionally because of a JMLException.

Inlining the annotations

Once the set-annotations at method specification level are generated, the next step is to inline them into the method bodies. To ensure that the appropriate set-statements are always executed at the end of the method body, the body is wrapped in a TryCatchFinally statement. The translation α_3 applies $\alpha_{3,\mathcal{C}}$ to all classes, which in turn applies $\alpha_{3,\mathcal{M}}$ to all methods in the class. This function generates one new local variable[8] result. The body of the method is changed as follows: all code is wrapped in two TryCatchFinally statements, to catch Throwable and RunTimeException exceptions[9]. In the *try* block, first pre_set is executed, followed by the body of the method. Then the result expression from the original body is evaluated, and assigned to result. Next, post_set is executed. Notice that the latter is only executed if the body actually terminates normally, otherwise the exception will simply be propagated. Finally, in the *catch* clauses, exc_set is executed. The new result expression of the method is the look up of the variable result. To conclude, pre_set, post_set and exc_set in the method specification are set to Skip. Figure 10 gives the formal definition of $\alpha_{3,\mathcal{M}}$ (where P is a program, and m a method).

To prove correctness of this translation, we use the following relation: all fields and ghost variables coincide, exceptions coincide, and all local variables that are declared in the original program coincide. In the correctness proof, we use that

[8] In fact, this should be a local *ghost* variable, but these are not yet supported by our formalisation, therefore we formalise it as a standard local variable.

[9] For simplicity, we do not model the exception hierarchy and thus TryCatchFinally can only catch a single exception, but in practice only one *try-catch-finally* instruction would be necessary.

the post_set and exc_set annotations do not throw any exceptions, and pre_set may only throw a JMLException. Moreover, we use that the set-annotations do not contain method calls, from which we can conclude that they do not modify any variables that are not explicitly mentioned in them. In particular, this allows to conclude that the new local variable is not changed by the set annotations.

5 Related Work

Security automata [Sch00] are widely used for monitoring security properties. The originality of our work lies in considering them as specifications in a general specification language, with the ultimate goal of static verification.

Closely related to our approach is work by Aktug *et al.* [AN08, Akt08, ADG08], who define a formal language for security policy specifications, ConSpec, that is similar to our PAs. They prove that a monitor can be inlined into the program's bytecode, by adding first-order logic annotations, and then they use a weakest precondition computation that essentially works the same as the annotation propagation algorithm that we plan to use [PBB+04] to produce a fully annotated, verifiable program. In contrast, our algorithm is defined for source code, and connects with the general-purpose specification language JML. This allows the use of JML verification tools, to verify the actual policy adherence. And of course, correctness of our inlining algorithm has been proven with a theorem prover.

Cheon and Perumendla propose an extension of JML to specify allowed sequences of methods calls in a regular expression-like notation [CP07]. This results in succinct specifications, but of limited expressiveness. Even our *limited SMS* example is out of their scope, because it contains a counter used only by the specification. Further, they only target run-time verification.

There are several tools that translate temporal properties into JML annotations: AutoJML [HOP03] translates finite state machine specifications into JML annotations and can also generate a code skeleton for a smart card applet; JAG [GG06] translates properties in (a subset of) temporal logic, including liveness properties. However, they typically do not distinguish between method entry and exit, and moreover, correctness of the translation algorithm has not been proven.

For more information about policy languages, monitor inlining and specifying policy adherence, we refer to Section 4.10 of Aktug's thesis [AN08].

6 Conclusions and Future Work

This chapter presents an algorithm to inline security automata, in the form of JML annotations. The translation is defined in several steps, thanks to the introduction of method-level set-annotations as extension to JML. All steps are formalised and proven correct, using the PVS theorem prover. The algorithm might seem trivial, but several subtleties complicate the proof, i.e. evaluating the

specifications in the right order, dealing with side effect-freeness of annotations and the possibility that a *finally* block hides exceptions.

First of all, for the main step, where the monitor is encoded in method-level set annotations, it is not straightforward to state the correct relation that is maintained between the two program executions. In particular, once the halted state is reached, the correspondence between the two program states is not preserved anymore, because the monitored program might continue executing, while the annotated program throws an exception. Also, to prove correctness of this translation step, it was important to specify the evaluation of the precondition and the postcondition, the class invariant and the pre_set and post_set annotations in the right order, to ensure that the relation was re-established as quickly as possible – so that the induction hypothesis could be applied for the other steps. Adding the additional assert to the pre_set annotation was also crucial for correctness. Finally, a last important issue here was the behaviour of the TryCatchFinally statement, where exceptions in the *finally* block can overwrite other exceptions.

The formalisation has been developed for a subset of JAVA. We believe that extending it to full (sequential) JAVA would be relatively straightforward. However, generalising to properties that are not restricted to a single class or that are related to multithreading might be more challenging. In particular, for monitoring properties of objects, we would need an aliasing analysis. Consider the property "f.open() *before* f.close()" (i.e., only opened files can be closed). A program f.open(); g.open(); f.close(); g.close(); should be accepted by the monitor, except when f and g are aliases.

The ultimate goal of our work is to statically verify adherence to security policies. To achieve this, a weakest precondition calculus can be used to generate preconditions and postconditions, based on the generated Set annotations. In earlier work, we presented such a propagation algorithm [PBB+04], and proved correctness for a limited case (instance variables and branches are not considered). It is future work to overcome these limitations.

Acknowledgements We thank Erik Poll for his useful comments on an earlier draft of this work, and Igor Siveroni, who started the work on this topic and came up with the idea to use method-level set-annotations.

Chapter 4

Reasoning about Assignments in Recursive Data Structures

Revised version of *Reasoning about Assignments in Recursive Data Structures.*
In Proceedings of the 13th Brazilian Symposium on Formal Methods (SBMF 2010).
Natal, Brazil, November 8–12, 2010.

Reasoning about Assignments in Recursive Data Structures

Alejandro Tamalet and Ken Madlener

Institute for Computing and Information Sciences (iCIS),
Radboud University Nijmegen, The Netherlands

Abstract. Proving properties of programs involving mutable data structures is a challenging enterprise. Because of aliasing, an assignment can modify the value of a variable not explicitly mentioned. Therefore it is important to be able to say what has changed after an assignment, and how. This chapter presents a framework to reason about the effects of assignments in recursive data structures. We define an operational semantics for a core language based on Meyer's ideas for a semantics for the object-oriented language EIFFEL. A series of field accesses, e.g. $\mathtt{f}_1 \bullet \mathtt{f}_2 \bullet \ldots \bullet \mathtt{f}_n$, can be seen as a path in the heap. We provide rules that describe how these *multidot* expressions are affected by an assignment. Using multidot expressions to construct an abstraction of a list, we show the correctness of a list reversal algorithm. This approach does not require induction and the reasoning about the assignments is encapsulated in the mentioned rules. We also discuss how to use this approach when working with other data structures and how it compares to the inductive approach. The framework, rules and examples have been formalised and proven correct using the PVS proof assistant.

1 Introduction

In order to verify pointer programs that manipulate recursive data structures, one generally identifies the pointer structure embedded in the heap with an abstract model. A concrete instance is a mapping of a set of objects in the heap connected by a field such as `next` to an abstract list of objects. The mapping is called the *abstraction* and the abstract list is called the *abstract model*. An operation performed by the program on a pointer structure in the heap has a corresponding operation on the abstract model. For example, the operations performed by a list reversal algorithm have the combined effect that the abstract list is reversed at the end of the execution. The standard way to define data abstractions is by recursion on the structure of (the data type of) the abstract model.

Verification of pointer programs is a non-trivial task due to the possibility of aliasing. Modifying data through one name implicitly modifies the values associated to all aliased names. If two portions of the heap are disjoint, an assignment in one part of the heap does not affect the other; this is called *local reasoning*.

Local reasoning is essential for scalability and several approaches to obtain it have been studied, see e.g. Separation Logic [Rey02] and Region Logic [RBN10].

When it is not known how the heap is partitioned or when working within a region that may contain aliases, we have to reason about how a change to (a portion of) the heap affects the corresponding abstract model. This complements local reasoning. In this chapter we focus on the effects of assignments to abstract models. We present our work in the setting of a core language, inspired by Meyer's ideas for a semantics for the object-oriented language EIFFEL [Mey03]. However, we do not depend on any characteristic properties of EIFFEL; our language is general enough to serve as the basis for other languages.

Our framework allows us to express multidot field access expressions, multidot expressions for short, of the form $\mathtt{f}_1 \bullet \mathtt{f}_2 \bullet \ldots \bullet \mathtt{f}_n$. A multidot expression consisting of a series of `next`-fields describes a path from the head of a list to one of its elements. If we instantiate it with a series of `left` and `right`-fields we can describe the path from the root of a binary tree to any node or leaf. In general, a multidot expression describes a path in the heap where the elements are connected by field accesses.

The main contribution of this chapter is to provide a set of rules that precisely describe the value of a multidot expression after an assignment, and to show how these rules can be applied for verification of programs that manipulate recursive data structures. The given rules are categorised into separation rules, where the assignment has no effect on the multidot expression, and interference rules, where the assignment does have effect on the multidot expression. We have applied these rules to show the correctness of an in-place list reversal algorithm by mapping each element of the list to a multidot expression. The proof does not require induction; it is encapsulated in the separation and interference rules that are applied to reason about effects on the abstract list model. We also discuss how to apply the same principles to other recursive data structures and we make a comparison with the standard inductive approach. Our work has been completely carried out in the theorem prover PVS [OSRS01].

This chapter is organised as follows. Section 2 defines the language we shall work with. In Section 3 we present the rules that describe the effects an assignment can have on a multidot expression and in Section 4 we apply these rules to prove the correctness of a list reversal algorithm and we discuss the applicability to other data structures. We compare the approach described in this chapter with the standard inductive approach and we give pointers for future work in Section 5. Related work is discussed in Section 6 and conclusions are drawn in Section 7.

2 The Model

This section describes an operational semantics of a core object-oriented language. The focus is on the features needed to understand the properties discussed in the next section, i.e., we do not model some typical object-oriented features

like inheritance. Since we have a shallow embedding, other features could be added independently. The interested reader can find the full PVS formalisation at `http://cs.ru.nl/~tamalet`. For a short introduction to PVS see Section 2 in the introductory chapter.

2.1 The Heap

In our model we consider all values to be an object or void. The set `Object` is defined as an uninterpreted type that represents non-void objects. Instances of $\texttt{Object}_\texttt{v}$ have the possibility of being `void`:

$$\texttt{Object}_\texttt{v} : \textsc{type} = \{\texttt{obj}(\texttt{obj} : \texttt{Object}),\ \texttt{void}\}$$

A basic approach to model the heap, due to Burstall [Bur72] and more recently emphasised by Bornat [Bor00], is to model it as a collection of functions of type $\texttt{Object} \to \texttt{Object}_\texttt{v}$, one for each class field (i.e. the component). This is sometimes called the component-as-array model [FM07, HM07]. This modelling encodes the fact that changing to what object a field points to does not affect other fields. This has the important consequence that whenever one field is updated, we do not need to propagate that update to the other fields. Since a field is determined by the name of the function that models it, we obtain a "separation by syntax".

Our heap is a grouping of field functions, indexed by their field names:

$$\texttt{Heap} : \textsc{type} = [\texttt{Name} \to [\texttt{Object} \to \texttt{Object}_\texttt{v}]]$$

where `Name` is a set representing the field names. Given a heap `h` and a field name `f`, $\texttt{h}(\texttt{f})$ is the corresponding field function. This indexing allows us to reason about field names, which is not possible when using a loose set of field functions as in the component-as-array model. There, the names of the fields are fixed by the names of the functions that model them while in our model the names are parametric: $\texttt{h}(\texttt{f})$ and $\texttt{h}(\texttt{g})$ represent different fields if and only if $\texttt{f} \neq \texttt{g}$, while in the component-as-array model two functions with different names like necessarily represent different fields. We use this to express meta-properties about multidot field expressions in Section 3. The separation by syntax provided by the component-as-array model is lost in this model, because a field update is now an update of the heap function. With the meta-level properties presented in the rest of this chapter, we obtain a reincarnation of separation by syntax.

The above definition of the heap highlights the relationship with the component-as-array model. However, defining the heap as a function of type $[\texttt{Object} \to [\texttt{Name} \to \texttt{Object}_\texttt{v}]]$ may seem more intuitive. In this definition we first fix an object and then we ask for a field name to obtain its value. As the functions are total (required by PVS), both definitions are in fact equivalent. This means that every field should be defined at every object. This is of course not realistic, however, accesses to undefined fields can be handled by a preliminary static analysis.

2.2 Expressions, Statements and Compositions

We model expressions, statements and their compositions following Meyer's ideas for a semantics for EIFFEL [Mey03]. A distinctive aspect of this approach is that expressions and statements are evaluated relative to an object, which is provided together with the heap as argument.

We deal with null-pointer dereferencing in language constructs, as opposed to avoiding it by type constraints. In our experience, the second approach leads to cumbersome specifications because the result of each expression and statement must be checked for definedness before composing them.

There are two syntactic categories: expressions (without side effects) and statements:

```
Expr : TYPE = {e : [Object_v⊥ , Heap⊥ → Object_v⊥] |
  ∀ (o : Object_v⊥ , h : Heap⊥) :
    bottom_or_void?(o, h) ⇒ bottom?(e(o, h))}

Stmt : TYPE = {S : [Object_v⊥ , Heap⊥ → Heap⊥] |
  ∀ (o : Object_v⊥ , h : Heap⊥) :
    bottom_or_void?(o, h) ⇒ bottom?(S(o, h))}
```

To define a semantics for EIFFEL, Meyer works with partial functions [Mey03]. In most theorem provers, including PVS, functions have to be total. For this reason we use lifted arguments, to represent undefinedness. The `bottom_or_void?(o, h)` predicate returns `true` if and only if `o` is undefined or void or `h` is undefined. By using predicate subtypes, we ensure that whenever an expression or statement is evaluated in `void` or in an undefined object or state, the result is undefined. This shifts checking for void or bottom from the specification to type correctness obligations that PVS generates automatically.

The expression `Current` (called `this` or `self` in some languages) returns the current object:

```
Current : Expr = λ (o : Object_v⊥ , h : Heap⊥) :
  IF bottom_or_void?(o, h) THEN bottom ELSE o
```

The operators • and ; compose expressions and statements. If `x` is an expression, `S` an statement and `r` is either of them, we define:

```
S ; r = λ (o : Object_v⊥ , h : Heap⊥) : r(o, S(o, h))
x • r = λ (o : Object_v⊥ , h : Heap⊥) : r(x(o, h), h)
```

The normal uses are state compositions `S;T` and field access `x • y`. The overloading allows us also to write `S; x`, which returns the value of evaluating `x` after the statement `S`, and `x • S`, which can be thought as a qualified call of `S` from `x`.

We define in PVS an automatic conversion that translates a name `f` into its corresponding field function, i.e. to the expression

```
λ (o : Object_v⊥ , h : Heap⊥): h(f)(o)
```

whenever needed. This allows us to express a field access directly as `x • f`. We also define a conversion that translates an `o : Object` into `obj(o) : Object_v`, and one that translates an `o : Object_v` into `up(o) : Object_v⊥`, to reduce the amount of syntax.

`IF`-statements are mapped to IF-expressions in the logic of PVS. For reasons of succinctness we omit the treatment of `WHILE`.

As the reader has probably noticed, expressions do not return a new state and thus the model does not handle side effects. This is a limitation inherited from Meyer's work. It could be avoided by making expressions and statements return both an object and a heap, and adapting the compositions. However, the effect of side effects can be mitigated by working directly with the object that results from an expression instead, as shown next.

2.3 Assignments

At its core, an assignment is an update of a heap-function in a particular point (consisting of a field and an object):

```
update(f : Name, p : Object, h : Heap, q : Object_v) : State =
  λ (g : Name)(o : Object) :
    IF p = o ∧ f = g THEN q ELSE h(g)(o)
```

Our model forces us to explicitly deal with undefinedness due to dereferencing of void. The `update` operation is encapsulated in an operator := that assigns an object `q` to the field `f` of the object `p` in the heap `h`:[1]

```
:= (f : Name, q : Object_v) : Stmt =
  λ (p : Object_v⊥, h : Heap⊥) :
    IF bottom_or_void?(p) ∨ bottom?(h) THEN bottom
    ELSE update(f, obj(down(p)), down(h), q)
```

If the assignment is made in an undefined state or tries to assign to void, the error is propagated. This is required by the definition of `Stmt`. The next step is to define local assignments `f := e` and qualified assignments $e_1 \bullet f := e_2$.

```
:= (f : Name, e : Expr) : Stmt =
  λ (o : Object_v⊥, h : Heap⊥) :
    IF bottom?(e(o, h)) THEN bottom
    ELSE (f := down(e(o, h)))(o, h)

:= (e_1 : Expr, f : Name, e_2 : Expr) : Stmt =
  λ (p : Object_v⊥, h : Heap⊥) :
    IF bottom?(e_2(p, h)) THEN bottom
    ELSE (f := down(e_2(p, h)))(e_1(p, h), h)
```

[1] In the PVS formalisation we have called this function <=, because := is reserved.

These definitions are not relevant for the development of this chapter and we hence we focus on assignments of objects. We shall use the above variable names throughout the rest of this chapter.

An assignment affects a field access if and only if the object where the field is evaluated is the one where the assignment was made and the field being accessed is the one that was assigned to. This is summarised in the following two basic separation and interference properties (both assume that $\mathtt{o}, \mathtt{h}$ is not bottom or void):

Property 1. If $\mathtt{p} \neq \mathtt{o}$ or $\mathtt{f} \neq \mathtt{g}$, then $\mathtt{g}(\mathtt{o}, (\mathtt{f} := \mathtt{q})(\mathtt{p}, \mathtt{h})) = \mathtt{g}(\mathtt{o}, \mathtt{h})$.

Property 2. If $\mathtt{p} = \mathtt{o}$ and $\mathtt{f} = \mathtt{g}$, then $\mathtt{g}(\mathtt{o}, (\mathtt{f} := \mathtt{q})(\mathtt{p}, \mathtt{h})) = \mathtt{q}$.

The proofs of these two properties amount to expanding the definition of := and applying several case-splits. When the assignment is replaced with a qualified assignment $\mathtt{e}_1 \bullet \mathtt{f} := \mathtt{e}_2$, then analogous properties hold, but $\mathtt{p} = \mathtt{o}$ is replaced by $\mathtt{e}_1(\mathtt{o}, \mathtt{h}) = \mathtt{o}$.

One has to explicitly apply properties 1 and 2 as proof steps to reason about the effect of an assignment in the presented semantics. The key condition is $\mathtt{p} = \mathtt{o} \wedge \mathtt{f} = \mathtt{g}$. The latter is a syntactical comparison and thus can be done automatically. However, most of the time comparison between objects cannot be discharged automatically, unless we have information about the layout of the heap, see Section 5.

An interesting discrepancy between Meyer's work and ours concerns Meyer's T24 property [Mey03] about *relative assignments*, which says that

$$\mathtt{f} \bullet (\mathtt{g} := \mathtt{e}) \ ; \mathtt{f} \bullet \mathtt{g} = \mathtt{e}$$

Besides the typo that the right-hand side should actually be $\mathtt{f} \bullet \mathtt{e}$ instead of just $\mathtt{e}$ (because $\mathtt{e}$ is evaluated after accessing $\mathtt{f}$), the theorem prover has reminded us of the degenerate cases where (1) the assignment fails or (2) $\mathtt{f} = \mathtt{g}$ and $\mathtt{f}(\mathtt{o}, \mathtt{h})$ points back to the current object. This resulted in the following property:

```
meyer_T24 : LEMMA
  f • (g := e) ; (f • g) =
    λ (o : Object_v⊥, h : Heap⊥) :
      IF bottom?((f • (g := e))(o, h))
      THEN bottom
      ELSIF f = g∧f(o, h) = o
      THEN (e • g)(o, h)
      ELSE (f • e)(o, h)
```

3 The Effect of Assignments on Multidot Expressions

In this section we look at expressions of the form

$$(\mathtt{g}_1 \bullet \ldots \bullet \mathtt{g}_n)(\mathtt{o}, (\mathtt{f} := \mathtt{q})(\mathtt{p}, \mathtt{h})), \tag{1}$$

where the $\mathtt{g_i}$ and $\mathtt{f}$ are field names, $\mathtt{o}$ and $\mathtt{p}$ are $\mathtt{Object_{v\perp}}$ and $\mathtt{q}$ is of type $\mathtt{Object_v}$. Because undefinedness due to dereferencing void is not an essential part of the discussion, we shall omit it in the rest of the chapter.

Properties 1 and 2 describe the result of a very simple multidot, namely one where $\mathtt{n}$ is equal to 1. There, the condition which determines the result is $\mathtt{p} = \mathtt{o} \wedge \mathtt{f} = \mathtt{g}$. In multidot field expressions of arbitrary length a similar condition determines the result, but now it must be taken into account that there can be several places in the path from $\mathtt{o}$ to $(\mathtt{g_1} \bullet \ldots \bullet \mathtt{g_n})(\mathtt{o}, (\mathtt{f} := \mathtt{q})(\mathtt{p}, \mathtt{h}))$ such that $\mathtt{p}$ is the origin and the field name is $\mathtt{f}$. Thus we are interested in the set of indexes $\mathtt{k}$ such that:

$$\mathtt{p} = (\mathtt{g_1} \bullet \ldots \bullet \mathtt{g_{k-1}})(\mathtt{o}, \mathtt{h}) \text{ and } \mathtt{f} = \mathtt{g_k}\,.$$

The properties we present in this section are categorised into *separation rules*, where the assignment has no effect on the multidot field expression, and *interference rules*, where the assignment does have effect on the multidot field expression. Moreover, we now have a choice to look at the heap $\mathtt{h}$ before the assignment, or at the heap $\mathtt{h'} = (\mathtt{f} := \mathtt{q})(\mathtt{p}, \mathtt{h})$ after the assignment. For the separation properties this does not make a difference, but for the interference properties it does.

The properties we derive about multidot expressions in this section are at the meta-level. Although it is possible to use them to reason about a particular multidot in a program, the intended use is to reason about the effects of assignments on recursive data structures. Examples that demonstrate the application are given in Section 4.

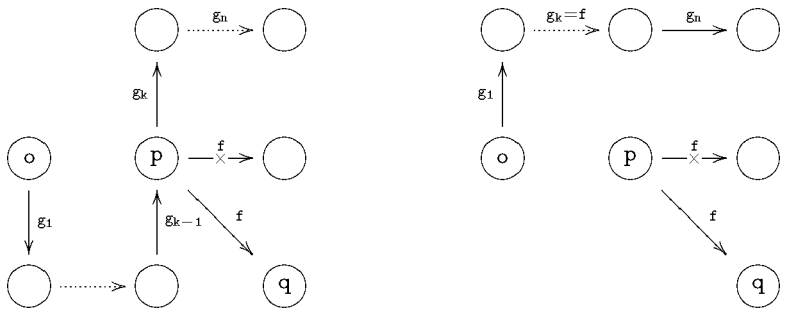
To improve readability, the notation for multidot expressions differs from the actual syntax used in PVS. In the last subsection we show the concrete PVS formalisation of a property. We will use graphs representing a portion of the heap $\mathtt{h'}$ to show examples of the properties. In these graphs nodes are objects and edges are labelled with an attribute name. An edge $\xrightarrow{\mathtt{f}}$ from an object $\mathtt{o}$ to an object $\mathtt{p}$ means that $\mathtt{f}(\mathtt{o}, \mathtt{h'}) = \mathtt{p}$. The edge removed by the heap update is depicted as $\overset{\mathtt{f}}{\times\!\!\rightarrow}$.

3.1 Looking at the Heap Before the Assignment

The assignment in (1) may or may not modify the multidot field expression. Graphically, what matters is whether the edge that has changed belongs to the path followed by the multidot field expression or not. A particular edge is determined by its object of origin and the field name. Hence, the condition that determines whether the assignment influences the multidot expression is whether or not the following set is empty:

$$\mathtt{K_{pre}} \;\widehat{=}\; \{\, \mathtt{k} : \mathtt{nat} \mid \mathtt{k} < \mathtt{n} \wedge \mathtt{p} = (\mathtt{g_1} \bullet \ldots \bullet \mathtt{g_{k-1}})(\mathtt{o}, \mathtt{h}) \wedge \mathtt{f} = \mathtt{g_k} \,\}.$$

We start with the case where $\mathtt{K_{pre}}$ is empty, i.e., the edge changed by the assignment is not part of the multidot expression, as shown in Figure 1.
As the edge that changed was not part of the multidot expression, the assignment does not have an effect on it.



(a) $\mathtt{p} = (\mathtt{g}_1 \bullet \ldots \bullet \mathtt{g}_{k-1})(\mathtt{o},\mathtt{h})$ but $\mathtt{f} \neq \mathtt{g}_k$. (b) $\mathtt{f} = \mathtt{g}_k$ but $\mathtt{p} \neq (\mathtt{g}_1 \bullet \ldots \bullet \mathtt{g}_{k-1})(\mathtt{o},\mathtt{h})$.

Fig. 1. Examples where $\mathtt{K}_{\mathtt{pre}}$ is empty.

Property 3. ($\mathtt{empty\_K}_{\mathtt{pre}}$) If $\mathtt{K}_{\mathtt{pre}}$ is empty, then

$$(\mathtt{g}_1 \bullet \ldots \bullet \mathtt{g}_n)(\mathtt{o}, \mathtt{h}') = (\mathtt{g}_1 \bullet \ldots \bullet \mathtt{g}_n)(\mathtt{o}, \mathtt{h})\,.$$

Now consider the case where $\mathtt{K}_{\mathtt{pre}}$ is not empty. Figure 2 depicts an example with two indexes $\mathtt{i}$ and $\mathtt{k}$ in $\mathtt{K}_{\mathtt{pre}}$ such that $\mathtt{k} < \mathtt{i}$. If there are several indexes in

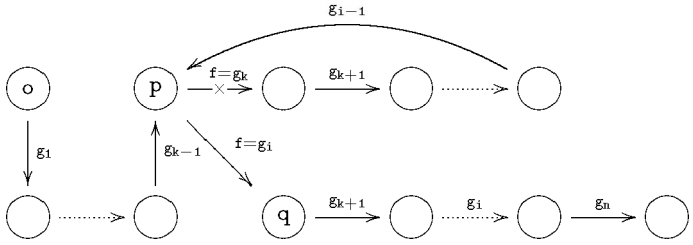


Fig. 2. Example with two indexes $\mathtt{k} < \mathtt{i}$ in $\mathtt{K}_{\mathtt{pre}}$.

$\mathtt{K}_{\mathtt{pre}}$, it means that there are several loops starting at $\mathtt{p}$. The assignment breaks the first edge in these loops. In the heap after the assignment, the edge that joins $\mathtt{p}$ with $\mathtt{q}$ is determined by the *least* element in $\mathtt{K}_{\mathtt{pre}}$.

Property 4. ($\mathtt{min\_K}_{\mathtt{pre}}$) If $\mathtt{k} = \mathtt{min}(\mathtt{K}_{\mathtt{pre}})$, then

$$(\mathtt{g}_1 \bullet \ldots \bullet \mathtt{g}_n)(\mathtt{o}, \mathtt{h}') = (\mathtt{g}_{k+1} \bullet \ldots \bullet \mathtt{g}_n)(\mathtt{q}, \mathtt{h}')\,.$$

Since the assignment may also affect the path that goes from $\mathtt{q}$ to the final value, the right hand side must still be evaluated in $\mathtt{h}'$.

3.2 Looking at the Heap After the Assignment

Instead of looking at when the multidot expression follows the edge that changed in $\mathtt{h}$, we will now look at when it follows the new edge in $\mathtt{h}'$. That is, we will look at the set:

$$\mathtt{K_{pos}} \;\widehat{=}\; \{\, \mathtt{k} : \mathtt{nat} \mid \mathtt{k} < \mathtt{n} \wedge \mathtt{p} = (\mathtt{g}_1 \bullet \ldots \bullet \mathtt{g}_{k-1})(\mathtt{o}, \mathtt{h}') \wedge \mathtt{f} = \mathtt{g}_k \,\}.$$

As expected, if the new edge is never traversed, the multidot expression does not change.

Property 5. ($\mathtt{empty\_K_{pos}}$) If $\mathtt{K_{pos}}$ is empty, then

$$(\mathtt{g}_1 \bullet \ldots \bullet \mathtt{g}_n)(\mathtt{o}, \mathtt{h}') = (\mathtt{g}_1 \bullet \ldots \bullet \mathtt{g}_n)(\mathtt{o}, \mathtt{h})\,.$$

Now assume that there is at least one index in $\mathtt{K_{pos}}$. In Figure 3 we see an example with two such indexes $\mathtt{i}$ and $\mathtt{k}$ with $\mathtt{i} < \mathtt{k}$. In this case the result of

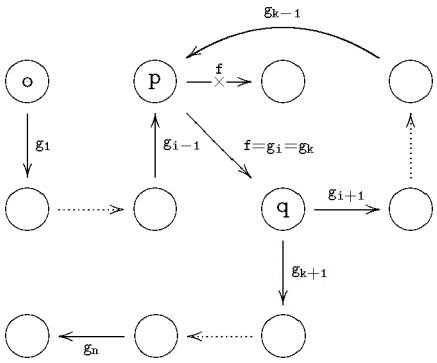


Fig. 3. Example with two indexes $\mathtt{i} < \mathtt{k}$ in $\mathtt{K_{pos}}$.

the multidot expression can be described as either $(\mathtt{g}_{k+1} \bullet \ldots \bullet \mathtt{g}_n)(\mathtt{q}, \mathtt{h}')$ or as $(\mathtt{g}_{i+1} \bullet \ldots \bullet \mathtt{g}_k \bullet \mathtt{g}_{k+1} \bullet \ldots \bullet \mathtt{g}_n)(\mathtt{q}, \mathtt{h}')$. If we take the greatest index in $\mathtt{K_{pos}}$, we get the shortest path to the resulting value and since the rest of the edges are not affected by the assignment we can describe the result in terms of $\mathtt{h}$. This is expressed in the following properties.

Property 6. ($\mathtt{forall\_K_{pos}}$) For all $\mathtt{k}$ in $\mathtt{K_{pos}}$,

$$(\mathtt{g}_1 \bullet \ldots \bullet \mathtt{g}_n)(\mathtt{o}, \mathtt{h}') = (\mathtt{g}_{k+1} \bullet \ldots \bullet \mathtt{g}_n)(\mathtt{q}, \mathtt{h}')\,.$$

Property 7. ($\mathtt{max\_K_{pos}}$) If $\mathtt{k} = \mathtt{max}(\mathtt{K_{pos}})$, then

$$(\mathtt{g}_1 \bullet \ldots \bullet \mathtt{g}_n)(\mathtt{o}, \mathtt{h}') = (\mathtt{g}_{k+1} \bullet \ldots \bullet \mathtt{g}_n)(\mathtt{q}, \mathtt{h})\,.$$

3.3 PVS formalisation

Given a list of names `fs`, the dot composition of the corresponding attributes is formalised as

```
multidot(fs : list[Name]) : RECURSIVE Expr =
  IF null?(fs) THEN Current
  ELSIF null?(cdr(fs)) THEN car(fs)
  ELSE car(fs) • multidot(cdr(fs))
MEASURE length(fs)
```

Note that because `e • Current` = `e` does not hold when `e` evaluates to void, we cannot simply append `Current` at the end of the multidot expression.

As an example of the PVS formalisation, we show a property that combines $\texttt{empty\_K}_{\texttt{pos}}$ and $\texttt{max\_K}_{\texttt{pos}}$ in a property at the source code level. Since it is written as an equality between functions, it can be used as a rewrite rule.

```
multidot_after_assignment_pos : LEMMA
  ∀ (f : Name, gs : list[Name], x, e : Expr,
        o : Object_v⊥, h : Heap⊥) :
    ((x • f := e ; multidot(gs))(o, h) =
      LET h' = (x • f := e)(o, h),
          K_pos = λ (k: below[length(hs)]) :
                    x(o, h) = multidot(take(gs, k))(o, h') ∧
                    f = nth(gs, k) IN
      IF bottom?(h') THEN bottom
      ELSIF empty?(K_pos) THEN multidot(gs)(o, h)
      ELSE LET k = max(K_pos) IN
        IF k = length(gs) - 1 THEN e(o, h)
        ELSE (e • multidot(drop(gs, k+1)))(o, h)
```

This property describes in terms of `h` all the possible outcomes of `multidot(gs)` when evaluated in $\texttt{h}'$. The (finite) set $\texttt{K}_{\texttt{pos}}$, which is described by its characteristic function, is the set of indexes satisfying (3.2). If the assignment resulted in an error then the result is an error. If $\texttt{K}_{\texttt{pos}}$ is empty then the multidot expression is unchanged. Otherwise, let `k` be the greatest element in $\texttt{K}_{\texttt{pos}}$. The result is then as stated in $\texttt{max\_K}_{\texttt{pos}}$ (with a shift of indexes due to lists starting at 0 in PVS). But again because `e • Current` is not equal to `e` when evaluated on void, we have to make a special case for when the multidot expression ends exactly at `e`. There is a similar lemma that combines $\texttt{empty\_K}_{\texttt{pre}}$ and $\texttt{min\_K}_{\texttt{pre}}$.

The intuitive way to prove these properties is by structural induction on `gs`. One would like to reason about the last edge of the multidot expression and use the inductive hypothesis on the path that leads to it. The problem with this approach is that on the non-empty case we have to reason about a list of the form `cons(g, gs)`. Therefore, we get to reason about the first edge, not the last one. To overcome this problem we defined a function `multidot_rev` that chains the arguments in the reverse order. Then we proved the corresponding lemmas that

work with the reversed list by induction on `gs`. Finally, the original lemmas were proven using their reversed counterpart by instantiating `gs` with `reverse(gs)`.

4 Linearised Abstractions

In this section we look at examples of abstract models expressed in terms of multidot field expressions. We call this style of specifying *linearised*, because it is not by recursion on the structure of the abstract model. The properties derived in the previous section provide us a set of tools to reason about the effects of an assignment to a linearised abstraction.

4.1 Paths

The following definition abstracts a path embedded in the heap to a list `l` of `Objects`. The `i`th object in `l` is the object on the heap that can be accessed by requesting the first `i` fields describing the path.

```
Path(gs : list[Name], l : list[Object])
    (o : Object_v⊥, h : Heap⊥) : bool =
  length(gs) + 1 = length(l) ∧
  ∀ (i : below[length(l)]) :
    multidot(take(gs, i))(o, h) = nth(l, i)
```

Due to the possibility of undefinedness, we define the abstractions as predicates about the heap and the current object rather than as functions because in PVS functions must be total.

With the use of the spatial separation lemmas for multidot expressions we can prove the following separation lemma for paths (recall that $\mathtt{h}' = (\mathtt{f} := \mathtt{q})(\mathtt{p}, \mathtt{h})$):

Property 8. If for all $\mathtt{i} < \mathtt{length}(\mathtt{l})$ it holds that $\mathtt{p} \neq \mathtt{nth}(\mathtt{l}, \mathtt{i})$ or $\mathtt{f} \neq \mathtt{nth}(\mathtt{gs}, \mathtt{i})$, and $\neg\, \mathtt{bottom?}(\mathtt{f}(\mathtt{p}, \mathtt{h}))$, then

$$\mathtt{Path}(\mathtt{gs}, \mathtt{l})(\mathtt{o}, \mathtt{h}') = \mathtt{Path}(\mathtt{gs}, \mathtt{l})(\mathtt{o}, \mathtt{h}).$$

Thinking again in terms of graphs, this lemma says that if an edge outside the path is modified, then the path is not affected by the assignment. To give an idea of how the multidot rules are applied, we sketch the proof of this lemma.

Proof sketch. We are supposed to show that the `Path` predicates are logically equivalent. In expanded form, we have to show that the following predicates are equivalent:

$$\forall\ (\mathtt{i}_1 : \mathtt{below[length(l)]}) : (\mathtt{g}_1 \bullet \ldots \bullet \mathtt{g}_{\mathtt{i}_1})(\mathtt{o}, \mathtt{h}') = \mathtt{nth}(\mathtt{l}, \mathtt{i}_1) \tag{2}$$

$$\forall\ (\mathtt{i}_2 : \mathtt{below[length(l)]}) : (\mathtt{g}_1 \bullet \ldots \bullet \mathtt{g}_{\mathtt{i}_2})(\mathtt{o}, \mathtt{h}) = \mathtt{nth}(\mathtt{l}, \mathtt{i}_2) \tag{3}$$

To show that (2) implies (3), we instantiate $\mathtt{i}_1$ with $\mathtt{i}_2$ and we apply $\mathtt{empty\_K}_{\mathtt{pos}}$. Then we have to show that $\mathtt{K}_{\mathtt{pos}}$ is indeed empty. If this was not the case then there would be a k such that

$$\mathtt{p} = (\mathtt{g}_1 \bullet \ldots \bullet \mathtt{g}_{\mathtt{i}_k})(\mathtt{o}, \mathtt{h}') = \mathtt{nth}(\mathtt{l}, \mathtt{k}) \quad \text{and} \quad \mathtt{f} = \mathtt{g}_{\mathtt{k}},$$

which is a contradicts the assumption that p is not in l. For the converse direction, we apply $\mathtt{empty\_K}_{\mathtt{pre}}$ in an analogous way. □

The interference property for paths describes how a path ending in p can be joined with a path beginning at q:

Property 9. If $\mathtt{p} \notin \mathtt{l}_0 +\!\!+ \mathtt{q} +\!\!+ \mathtt{l}_1$ and $\mathtt{c} = \mathtt{car}(\mathtt{l}_0 +\!\!+ \mathtt{p})$, then

$$\begin{aligned}&\mathtt{Path}(\mathtt{gs}_1 +\!\!+ \mathtt{f} +\!\!+ \mathtt{gs}_2,\ \mathtt{l}_0 +\!\!+ \mathtt{p} +\!\!+ \mathtt{q} +\!\!+ \mathtt{l}_1)(\mathtt{c},\ \mathtt{h}') = \\ &(\mathtt{Path}(\mathtt{gs}_1,\ \mathtt{l}_0 +\!\!+ \mathtt{p})(\mathtt{c},\ \mathtt{h}) \ \wedge\ \mathtt{Path}(\mathtt{gs}_2,\ \mathtt{q} +\!\!+ \mathtt{l}_1)(\mathtt{q},\ \mathtt{h}))\end{aligned}$$

The infix function ++ appends two lists. It is overloaded so that when one of its arguments is not a list, it is converted into a list with only that element. The proof uses the multidot rules $\mathtt{empty\_K}_{\mathtt{pos}}$ and $\mathtt{max\_K}_{\mathtt{pos}}$ for the implication from left to right and it uses the rules $\mathtt{empty\_K}_{\mathtt{pre}}$ and $\mathtt{min\_K}_{\mathtt{pre}}$ from right to left.

An important point about the proofs using linearised abstractions is that the induction is encapsulated in the rules about multidot expressions; to prove the above properties, we did not apply induction.

4.2 Example: Verification of an In-place List Reversal Algorithm

The Path abstraction can be specialised by Path(g, l), which instantiates the regular Path with a list of g-fields. By requiring the last node of Path(next, l) to point to void, we obtain an abstraction for lists in the heap:

```
List(l : list[Object])
    (o : Object_v⊥ , h : Heap⊥) : bool =
  Path(next, l)(o, h) ∧
  IF cons?(l) THEN void?(next(last(l), h))
  ELSE void?(o)
```

Note that List(null)(o, h) is true if and only if void?(o) is true, i.e. an empty list is represented by void. Similar separation and interference properties as the ones for Path can be proved for List.

To prove the correctness of the annotated in-place list reversal algorithm listed in Figure 4, we use standard Hoare-style reasoning. The annotations have type $\mathtt{Asrt} : [\mathtt{Object}_{\mathtt{v}\perp}, \mathtt{Heap}_{\perp} \to \mathtt{bool}]$ and a Hoare triple has the following meaning for $\mathtt{P}, \mathtt{Q} : \mathtt{Asrt}$ and $\mathtt{S} : \mathtt{Stmt}$:

$$\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\} \ \widehat{=}\ \forall\, (\mathtt{o} : \mathtt{Object}_{\mathtt{v}\perp}, \mathtt{h} : \mathtt{Heap}_{\perp}) :\ \mathtt{P}(\mathtt{o}, \mathtt{h}) \Rightarrow \mathtt{Q}(\mathtt{o}, \mathtt{S}(\mathtt{o}, \mathtt{h}))\,.$$

As can be seen in Figure 4, the current object o and the updated heap S(o, h) distribute over the connectives. So, the actual work to verify the correctness

of the list reversal algorithm amounts to simplifying expressions of the form $(\mathtt{g} \bullet \mathtt{List(l)})(\mathtt{o}, (\mathtt{e}_1 \bullet \mathtt{f} := \mathtt{e}_2)(\mathtt{o}, \mathtt{h}))$. By expanding the definitions of dot and assignment, this can be brought into the form of $\mathtt{List(l)}(\mathtt{o}', (\mathtt{f} := \mathtt{q})(\mathtt{p}, \mathtt{h}'))$, on which the separation and interference rules for the `List` abstraction can be applied.

```
{ λ(o, h) : ¬bottom_or_void?(o, h) ∧ a • List(As)(o, h) }
b := void;
WHILE (λ(o, h) : ¬void?(a(o, h))) DO
{ λ(o, h) : ¬bottom_or_void?(o, h) ∧
   ∃(as, bs : list[Object]) :
     (a • List(as))(o, h) ∧ (b • List(bs))(o, h) ∧
     disjoint?(as, bs) ∧ append(reverse(as), bs) = reverse(As) }
  tmp := a;
  a := a • next;
  tmp • next := b;
  b := tmp;
OD
{ λ(o, h) : ¬bottom_or_void?(o, h) ∧ (b • List(reverse(As)))(o, h) }
```

Fig. 4. In-place list reversal.

4.3 Other Data Structures

The linearised specification approach exemplified in the previous two sections can also be applied to other recursive data structures. Consider for example binary trees that store a value in each node:

```
binary_tree[σ]: TYPE = {leaf, node(v : σ, l, r : binary_tree)}
```

It is straightforward to define a predicate

```
get_node(bt : binary_tree[σ], path : list[Name], v : σ) : bool
```

that says whether by traversing `bt` in the order specified by `path`, we arrive at `v`.

```
get_node(bt: binary_tree[T], gs: list[Name], x: T):
    RECURSIVE bool =
  CASES bt OF
    leaf : FALSE,
    node(v, l, r) : IF null?(path) THEN v = x
                    ELSIF car(path) = left THEN get_node(l, cdr(gs), x)
                    ELSIF car(path) = right THEN get_node(r, cdr(gs), x)
                    ELSE FALSE
  ENDCASES
MEASURE size(bt)
```

Basically `get_node` maps each constructor application to the corresponding field name. We can now describe a binary tree in the heap by mapping each of its nodes to a multidot field access:

```
binary_tree_abstraction(bt : binary_tree[Object])
                       (o : Object_v⊥, h : Heap⊥) : bool =
  ∀ (x : Object, path : list[Name]) :
    get_node(bt, path, x) ⇒
      multidot(path)(o, h) = x
```

From the properties about multidot expressions presented in Section 3 one can obtain separation and interference lemmas for binary trees.

The same ideas can be applied to other tree-like structures. First make a linearised abstraction of the data structure: obtain the path from the root to each of its elements and use that path to describe the pointer structure in terms of multidot expressions. Then use the properties of Section 3 when reasoning about assignments. Data structures with loops can also be specified, e.g., a circular list is just a path that starts and ends in the same object.

5 Evaluation and Future Work

A natural way to define abstractions is by means of recursion on the structure of the abstract model. We single out the work by Mehta and Nipkow that uses this approach to verify several pointer programs [MN05]. The advantage of using induction is that it is a familiar general-purpose method that is integrated in the theorem prover. Much work has been devoted to automate proofs by induction, in particular to heuristics to instantiate the inductive hypothesis, e.g. *rippling* [BBHI05]. In the inductive approach one still has to reason about the effect of the assignments to the data structure, whereas using the rules given in Section 3 the focus is on when to apply each rule and in finding the extrema of the K-sets, which requires an instantiation.

Our experience is that both approaches require a comparable amount of proof work. However, there is still work to be done on investigating specialised version of the assignment rules and on the integration with the theorem prover as tactics. For example, if we know that there is no loop on a multidot expression, as is the case in tree-like structures, then we also know that the K-sets are either empty or have only one element. This eliminates the need to find the minima or the maxima of these sets.

Because both approaches lead to definitions that are essentially equivalent, the same properties hold. Hence, our approach can be seen as a complement rather than a replacement of inductive reasoning.

The component-as-array modelling exploits the fact that class fields that occur in a program are known statically. For objects this obviously does not work, because it cannot be known statically which objects are aliased. Reasoning about assignments ultimately reduces to reasoning about object equality. Therefore,

this framework would benefit from knowledge about the layout of the memory. The separation rules are used to provide local reasoning, but they are not a primitive of the logic as the star conjunct is in Separation Logic [Rey02] (see also Section 6). Hubert and Marché [HM07] propose a static separation analysis and show how it can be integrated in the component-as-array modelling. They split the heap into regions that are inferred by the separation analysis and accordingly relabel the field names as a combination `f_r` of the old field name `f` and a region `r`. This could be integrated into our model, for example by redefining the heap as

$$\texttt{Heap} : \textsc{type} = [\texttt{Region}, \texttt{Name} \to [\texttt{Object} \to \texttt{Object}_{\texttt{v}}]]$$

When it is inferred that two objects `x` and `y` lie in separate regions, the comparison between them can be avoided and the separation lemmas can be applied automatically.

6 Related Work

A first version of some of the rules presented in Section 3 first appeared in Tamalet's Master's thesis [Tam06].

In the seminal work of Bornat [Bor00] and also in the work by Meyer on a semantics for Eiffel [Mey03], pointer structures on the heap are related with abstract models via repeated composition of field requests. This has been a source of inspiration for this chapter. Bornat and Meyer both define a sequence closure operator that repeatedly requests a series of (the same) fields, yielding the list of objects that is traversed in the heap. This is essentially the same as our `Path` abstraction of Section 4.2. In this chapter we have given a complete and formalised overview of the effects of assignments to arbitrary multidot field expressions. A treatment of the sequential operator in the context of Eiffel has been given in an unpublished work by Blanco and Castro [BP05], restricted to the case of lists.

A perhaps more natural way to define abstractions is by the use of recursion on the structure of the abstract model. Mehta and Nipkow [MN05] used this approach to verify the correctness of several pointer programs. We have compared the inductive approach and the linearised approach in Section 5.

Hoare and Jifeng [HJ99] introduce a framework for the formulation of assertions about objects and pointers based on trace model of graphs and process algebra. They use a graphical notation very similar to the one used in this chapter. However, their model uses graph transformations to describe the changes to the state whereas we use an operational semantics. This choice makes reasoning about assignments in their framework much more difficult.

Our rules about an assignment followed by a multidot are meta-level properties of the language. To enable this meta-level reasoning we introduced a function `multidot` that maps a list of `Name`s to a suitable expression, which is essentially a deep embedding of multidot expressions. The rules about multidot expressions

are a reflection of the properties 1 and 2. For an instructive paper on reflection with examples in PVS we refer to [vHPPR98].

In the previous chapter we showed a framework to translate properties written as security automata into program annotations that can be checked at runtime. The main difference with the framework presented in this Chapter is that the expressivity and level of effort required by the user. The framework of the previous chapter has a limited expressivity, but it only requires the user to express the intended property as an automaton. Conversely, this framework is much more expressive but also requires the user to construct a proof for the property.

Local Reasoning

Local reasoning is the key to scalability in formal verification of programs. The way the heap is modelled in our framework is based on the component-as-array modelling idea of Burstall [Bur72]. Refinements of this modelling have been used as the core of weakest pre-condition calculus-based tools such as Krakatoa for the verification of JAVA programs, and CADUCEUS for the verification of C programs [FM07, MPM05]. A separation analysis tailored to integration with the component-as-array modelling has been proposed by Hubert and Marché [HM07]. Future work on the integration of this analysis with our work has been discussed in Section 5.

A well-studied approach to obtain local reasoning is that of Separation Logic, proposed by Reynolds [Rey02], which can be seen as a radical refinement of Burstall's idea. In Separation Logic disjointness of portions of the heap is made explicit in the logic. Its frame rule allows one to reason about just the relevant portion of the heap that a piece of code manipulates and later augment it with the rest of the heap. So far, no concrete case studies on industrial software make use of Separation Logic, but there is ongoing research on its automation, see e.g. [DF10, BCO06]. An implementation of [BCO06] has been developed inside the theorem prover HOL by Tuerk [Tue09].

A related line of research is Region Logic, whose goal it is to preserve the local reasoning of Separation Logic, but without using non-standard semantics of Hoare-triples. See [RBN10] for recent work.

7 Conclusions

In this chapter we have presented a novel approach to reason about assignments in recursive data structures. We have shown how recursive pointer structures can be described in terms of paths obtained by a series of field accesses. We have provided a formal model of these paths as multidot expressions and we have proved a set of rules that describe how an assignment can affect them. Using these rules we have derived separation and interference lemmas for lists and verified an in-place list reversal algorithm. A complete formalisation of the presented work has been carried out in the PVS theorem prover. We have also

shown how to apply this approach to reason about other data structures and we have compared it with the standard inductive approach.

Acknowledgements The authors would like to thank Marko van Eekelen and Sjaak Smetsers for their insightful comments on a draft version of this work and the anonymous reviewers for their comments.

Part II

Resource Analysis of Programs

Chapter 5

Introduction to Resource Analysis

The aim of *resource analysis* is to determine (upper and/or lower) bounds on resource usage. A "resource" here is any quantifiable physical or virtual component of limited availability within a computer system. Typical resources studied by resource analysis are heap space, stack space, clock cycles or some measure used to prove termination or productivity of non-terminating processes. There is a lot of research on resource analysis that applies varied techniques and requires different levels of user interaction to obtain diverse results. But two approaches deserve our attention for their relevance and quality of results: *sized types* and *amortised cost analysis*. They will be discussed in Section 1 and Section 2, respectively. In Section 3 we overview other subfields and techniques.

1 Sized Types

Sized types are types in a type and effect systems whose annotations express some size. The size does not need to be a physical size, it can be a measure of any resource. In particular functions types describe a size relation between the input of a function and its output. The type systems described in the following chapters belong to this category.

An early use of sized types is Dornic, Jouvelot and Gifford's polymorphic type and effect system for checking (but not inferring) time costs in a higher-order strict functional language [DJG92]. In a time system, i.e., a type and effect system to measure execution time, effects approximate the number of computation steps needed to reduce an expression to normal form. Dornic et al. introduce polymorphism via a "polymorphic lambda", which allows them to make the cost of functions parametric with respect to types.

However, the time system of Dormic et al. has some important limitations. First, recursive functions are always assigned an unbounded cost. This is because the cost of a recursive function depends on the sizes of arguments which are

not captured in the time system. The absence of size information also severely limits the precision of the analysis of higher-order functions, since the costs cannot depend on the sizes of arguments. Second, the type system does not allow subeffecting, i.e. subsuming a cost by a larger one; this is needed, e.g. to be able to type a conditional with different costs in each branch.

Reistad and Gifford [RG94] extended the time system of Dornic et al. and presented an algorithm to infer sizes and times based on algebraic reconstruction of effects [JG91]. This system has been applied to aid dynamically scheduling in a functional language with built-in parallelisation. The time bound is compared to the cost of spawning a new thread to decide whether parallel computation would be an opportunity to improve the overall performance. There is a trade off between using an upper bound and unnecessarily spawning threads, and a lower bound where parallelisation opportunities may be missed. They use upper bounds.

They avoid the difficult task of inferring precise sizes and costs for recursive functions by providing some primitive functions with known size effects and costs, such *map* and *fold* for lists. For other functions with non-trivial costs or size effects there is a special unbounded size, *long*, which is used when no more precise value can be inferred by the constraint solving.

Providing known primitives rather than using a more complex method ensures that sufficiently precise analyses can be made to allow dynamic parallelisation of a useful range of programs with an analysis of modest complexity (the constraints produced are solved by a quadratic fixed-point algorithm). Moreover, in some situations where no static information is available about the size of a data structure, the analysis may still be able to obtain an execution time bound relative to the unknown size. The user can later add run-time parallelisation based on the actual size observed.

In 1996, Hughes, Pareto and Sabry [HPS96] presented a type system extended with size information for proving liveness properties of reactive systems, namely termination and productivity.

The term language considered is purely-functional, non-strict and higher-order with let-bound polymorphism, general recursion and algebraic data types. The sized type system distinguishes *data* values (e.g., naturals or finite lists) from *codata* values (e.g., streams): the size of a data value is an upper bound on the number of constructors, while for a codata value it is a *lower* bound. The data types are annotated with a subscript or superscript size annotation for data or codata, respectively. Size annotations are restricted to arithmetic expressions using constants (natural numbers), variables and addition, but not multiplication; this subset of arithmetic can be checked computationally using a Presburger arithmetic solver such as the Omega Calculator [Pug91]. Presburger arithmetic is the first-order logic theory of the natural numbers with addition. Because it omits multiplication, Presburger arithmetic is less expressive than Peano arithmetic. However, the Presburger fragment is decidable [Coo72] while, by Gödel's incompleteness theorem, Peano arithmetic is undecidable.

As an example of the use of Pareto's work, consider the declarations for finite lists and infinite lists (streams)

$$\textbf{data List}\ a = \textbf{Nil} \mid \textbf{Cons}\ a\ (\textbf{List}\ a)$$
$$\textbf{codata Strean}\ a = \textbf{MkStream}\ a\ (\textbf{Stream}\ a),$$

the corresponding sized types for constructors are:

$$\textbf{Nil} : \forall a.\ \textbf{List}_1\ a$$
$$\textbf{Cons} : \forall i.\forall a.\ a \to \textbf{List}_i\ a \to \textbf{List}_{i+1}\ a$$
$$\textbf{MkStream} : \forall i.\forall a.\ a \to \textbf{Stream}^i\ a \to \textbf{Stream}^{i+1}\ a.$$

The types of the constructors **Cons** and **MkStream** express size relations: the result has one more constructor than the argument. Note that **Nil** has size one (not zero) because the size is the number of constructors of the value. A similar approach to counting constructors is taken in Chapter 6 of this thesis.

Sized data types can be seen as infinite families of approximations indexed by the number of constructors, e.g., $\textbf{List}_0 \subseteq \textbf{List}_1 \subseteq \ldots$ A special annotation ω is used to denote the "limit" of these approximations, e.g., $\textbf{List}_\omega$ is the type of all lists. As in the system of Reistad and Gifford mentioned before, the size ordering induces a structural subtyping relation on sized types. Subtyping is used, for example, to assign a sized type to a conditional with expressions of different sizes in the two branches. The novelty of the type system is a typing rule for recursion that embodies a principle of induction on sizes and that guarantees termination of recursive functions (and dually, productivity of corecursive ones). For further details we refer the reader to Pareto's degree thesis [Par98].

Pareto et al.'s work was subsequently extended with effects approximating stack and heap costs for Embedded ML [HP99], a strict first-order functional language using regions to control memory usage. We will review this work in 3.3 where we discuss other analyses based on regions.

Chin and Khoo [CK01] addressed the problem of inferring rather than just checking sized types. This system extends the prior work in two regards. Firstly, they allow sizes to be expressed as general Presburger constraints (first-order logic formulae with linear arithmetic over the integers). And secondly, by presenting an algorithm that computes a size formula for a recursive function using the *transitive-closure operation* [KPRS96] of the Omega Calculator [Pug91] on constraints.

For example, the analysis of Chin and Khoo infers the following size information for the standard list append function[2]:

$$\begin{aligned} &\mathit{append} : \textbf{List}^m(a) \to \textbf{List}^n(a) \to \textbf{List}^l(a) \\ &\quad \textbf{s.t.}\ \ \textbf{size}\ m \geq 0 \wedge n \geq 0 \wedge l = m + n \\ &\qquad\ \ \textbf{inv}\ 0 \leq m^+ < m \wedge n^+ = n \end{aligned}$$

The size constraint expresses the dependency between input and output list sizes. The invariance constraint appears only on recursive functions. Here n and

[2] For consistency reasons, the notation has been adapted.

m are the argument sizes of the initial call to *append*, and n^+ and m^+ are the sizes of any recursive (inner) call to *append*. The constraint describes the call invariant for all nested recursive calls. Such information is useful also in termination analysis or in programming transformations such as partial evaluation. These constraints are reminiscent of the predicates used in Chapter 7 to describe piecewise polynomials.

The term language is a strict, higher-order functional notation with integers, booleans and lists. Data types are annotated with size variables and all size information is expressed by separate size constraints. Typing judgements have the form $\Gamma \vdash e : (\tau, \phi)$, where e is an expression, τ an annotated type and ϕ a Presburger formula expressing the relationship among the variables of τ.

The notion of size is specific to each data type: the size of a list is its length; the size of an integer is its value (negative sizes for negative integers); boolean values **True** and **False** have sizes 1 and 0, respectively. This allows to encode control flow information in size constraints. For example, consider the function testing a list for emptiness:

$$\mathit{null}\ xs = \textbf{case}\ xs\ \textbf{of}\ [\,] \to \textbf{True} \mid x : xs \to \textbf{False}.$$

The sized type inferred for *null* is

$$\mathit{null} : (\textbf{List}^n(a) \to \textbf{Bool}^c,\ (c = 1\ \wedge\ n = 0)\ \vee\ (c = 0\ \wedge\ n > 0)),$$

where the size c indicates which branch was taken: if c is 1 then the list is empty and the size is 0; if c is 0 then the list is non-empty and the size is positive.

One limitation is that the type system is not type polymorphic since no size information is captured for type variables. Another limitation is that it fails to infer sizes of values inside lists. For instance, in the typing

$$\mathit{tail} : (\textbf{List}^n(\textbf{Int}^i) \to \textbf{List}^m(\textbf{Int}^j),\ n = 1 + m),$$

there is no information about the size of the elements inside the list, i.e., there is no relation between i and j in the inferred type. In a subsequent work [CKX03] the authors propose an extension to the sized type system with collection constraints to address this problem. However, the extended constraints fall outside the capabilities of a Presburger solver.

Vasconcelos and Hammond developed automatic inferences for a sized type analysis that are capable of deriving cost equations for abstract time and heap consumption from unannotated program source expressions based on the inference of sized types for recursive, polymorphic, and higher-order programs [VH04]. But they leave unsolved the recurrence equations that are obtained. In his PhD thesis [Vas08], Pedro Vasconcelos uses abstract interpretation techniques to automatically infer linear approximations of the sizes of recursive data types and the stack and heap costs of recursive functions. By including user-defined sizes, it is possible to infer sizes for algorithms on non-linear data structures, such as binary trees. However, the run time of the inference is exponential in the program size. His thesis also corrects an error in the soundness proof provided by Chin and Khoo [CK01].

The Amortised Heap Space (AHA) project has studied output-on-input polynomial size dependencies, where the polynomials are not necessary monotonic. In [SvKvE07a], the authors designed a type system for a first-order functional language where each type is annotated with a polynomial size expression. It allows to type function definitions over lists where the size of the output depends on the sizes of the inputs, but not on their values. For instance, $\mathsf{append}\colon \mathsf{List}_n(a) \times \mathsf{List}_m(a) \to \mathsf{List}_{n+m}(a)$, whereas delete (which deletes, from a list, the first occurrence of an element if it exists) does not have a type in that system since it may or may not delete an element. They also developed a test-based annotation inference procedure for that system in [vKSvE07]. In [TSvE09], i.e., Chapter 6 of this thesis, the authors extend the type system to algebraic data types with user-defined size functions. A complementary work [SvET11], i.e., Chapter 7, describes a type system and inference procedure that can obtain non-monotonic polynomial bounds (and not only exact sizes). However, there is still no implementation of this system. One of the goals of the new CHARTER project is to transfer these size analysis to the world of imperative programs. Some of this work has already materialised as an analysis for loop bounds, see [SKvE10].

Abel [Abe06] extended higher-order sized types to allow higher-kinded types with embedded function spaces. Barthe et al. [BGR08] describe a type system that is precise enough to type *quicksort* as a non-size increasing function. These system are used to prove termination, but do not tackle resource consumption in general.

2 Amortised Cost Analysis

The term "amortisation" is used in the financial world to denote the payment of an obligation in a series of instalments. Amortised complexity analysis aims at obtaining bounds for the cost of a sequence of operations; it is sometimes possible to obtain better worst-case bounds by amortisation than by reasoning about the costs of individual operations. For example, it might be possible to obtain a worst-case bound of $O(n)$ for a sequence of n operations even if some of the individual operations cost more than $O(1)$.

The "physicist's method" for deriving amortised bounds starts by assigning a non-negative potential function to data. The *amortised cost* of an operation is then defined as the sum of the actual cost (e.g., time cost or heap cells allocated) plus the difference in potential incurred by the operation. The key idea is to choose the potential functions so as to facilitate computing the amortised cost, e.g., in such a way as to make the amortised costs constant. Provided the potential is always non-negative and initially zero, the accumulated amortised costs will be an upper-bound on the accumulated actual costs (see [Oka98], page 41).

The concept of amortised cost was first developed in the context of complexity analysis by Tarjan in 1985 [Tar85].

The Hofmann-Jost analysis The seminal work by Martin Hofmann and Steffen Jost [HJ03] presents a type-based analysis for heap space usage using amortisation. Instead of extending type judgements with effects as done in [DJG92, RG94, HP99], their analysis is based on annotating data types with weights representing the relative contribution of parts of a data structure to the overall heap usage (the potential associated with the data structure).

The language under analysis, called $\mathbf{LF}_\diamond$, is a first order functional notation with a strict semantics and algebraic data types including sums, products, booleans and lists. There are two kinds of pattern-matching for heap-allocated values: a deallocating or destructive **match** and a non-deallocating **match**′ . The heap cost is defined by a big-step operational semantics instrumented with the size of a free list of heap cells; the free list reduces at each constructor application and grows at each **match** (but not at **match**′).

The augmented typing judgements are of the form $\Gamma, k \vdash e : A, k'$, where Γ is the context, i.e., the type assumptions, e is an expression, A is an annotated type and k, k' are non-negative rational numbers representing the available potential before and after the evaluation of e. The annotations in A together with k and k' give both an upper bound on the initial heap space needed for the evaluation of e and a lower bound on the available heap space after evaluation. For example, the judgement

$$x : \mathbf{List}(\mathbf{List}(\mathbf{Bool}, 1), 2), 3 \vdash e : \mathbf{List}(\mathbf{Bool}, 4), 5$$

says that if x is a list of lists of booleans then e evaluates to a list of booleans. Furthermore, if $x = [l_1, \ldots, l_n]$ then a free list of size $3+2n+1\cdot\sum_i |l_i|$ is sufficient to evaluate e and if e evaluates to a list l then the freelist will have size at least $5 + 4|l|$. Here $|\cdot|$ denotes the length of a list.

From this example we can see that type annotations play a very different role here than in the sized type systems: in the system of Hofmann and Jost an annotation represents not a size, but the coefficient of the heap cost incurred by a part of a data structure. The upper bound on the initial free list is a function of the (unknown) sizes of the input and the lower bound on the final free list size is a function of the (unknown) size of the output. No input/output size relation is obtained.

The type system of Hofmann and Jost performs an amortised analysis of the size of the free list: the coefficients in types represent the *potential* associated with the data structures; the typing rules constrain the annotations so that the amortised costs for each expression are properly accounted for. For example, the typing rules for constructing and deconstructing a list node are:

$$\frac{n \geq \mathbf{SIZE}(A \otimes \mathbf{List}(A, k)) + k + n'}{\Gamma, x_h : A, x_t : \mathbf{List}(A, k), n \vdash \mathbf{Cons}(x_h, x_t) : \mathbf{List}(A, k), n'}\ \mathbf{LF}_\diamond\text{:Cons}$$

$$\frac{\Gamma, n \vdash e_1 : C, n' \qquad \Gamma, x_h : A, x : \mathbf{List}(A,k), n + \mathbf{SIZE}(A \otimes \mathbf{List}(A,k)) + k \vdash e_2 : C, n'}{\Gamma, x : \mathbf{List}(A,k), n \vdash \mathbf{match}\ x\ \mathbf{with} \mid \mathbf{Nil} \Rightarrow e_1 \mid \mathbf{Cons}(x_h, x_t) \Rightarrow e_2 : C, n'}$$

$\mathbf{LF}_\diamond$:List-Elim

The rule $\mathbf{LF}_\diamond$:Cons specifies that the available potential n must be at least the amortised cost of **Cons**, that is, the actual heap cells used (given by the **SIZE** function) plus the potential k associated with the list elements (because the list length is increased by one). The size function is assumed to be *additive* over type product (which is denoted by $\otimes$): $\mathbf{SIZE}(A \otimes C) = \mathbf{SIZE}(A) + \mathbf{SIZE}(C)$. Dually, $\mathbf{LF}_\diamond$:List-Elim specifies that the available potential at the **Cons** alternative increases by the amortised cost (because **match** deallocates the cells).

Hofmann and Jost presented an algorithm that automatically infers the type annotations. Since annotations represent coefficients of the potential function, the system can only derive heap bounds that are linear on the sizes of data structures. Their technique associates each program P with a system of linear inequalities $\mathcal{L}(P)$ such that the valid annotated type derivations for P correspond to the admissible solutions of $\mathcal{L}(P)$; these solutions can be obtained by a standard linear programming solvers [DT97]. The worst-case theoretical complexity for solving linear programs is polynomial; the variants of the Simplex algorithm used in solver implementations, although exponential in the worst-case, are quite efficient in practice. This compares favourably with previous sized type systems like [HPS96, HP99, CK01] where type checking alone requires checking validity of Presburger constraints with doubly-exponential worst-case time. It also has a performance advantage over the type systems considered in Part II of this thesis, which are undecidable on the worst-case [SvKvE07a, TSvE09, SvET11], however, these type systems deal with non-linear input/output relationships.

Even with the restriction of linear bounds, the type systems of Hofmann and Jost is expressive enough to obtain heap costs for many list processing functions including insertion algorithms such as insertion sort and in-place quicksort. This is possible because the language implements deallocation using destructive matching. Unlike the sized type analysis of Hughes and Pareto [HP99] and the ones studied later in this thesis, the amortised analysis deals with the irregular divide-and-conquer recursions by "splitting" the potentials between the two recursive calls. Hofmann and Jost also present good results for a binary tree traversal and report successful analysis of other textbook examples.

One limitation of the analysis of Hofmann and Jost is that the inferred type annotations are sometimes not sufficiently polymorphic: it can happen that two usages of a function f require two different annotations. Even if both annotations are compatible with the definition of f, only one of them can actually be assigned in $\mathbf{LF}_\diamond$ and hence every use of a function shares the same potentials. Consider, for instance, the identity function $f : \mathbf{List}(\mathbf{Bool}) \to \mathbf{List}(\mathbf{Bool})$ applied to a list of booleans; if a particular use requires the annotation f :

$\mathbf{List}(\mathbf{Bool}, 5), 3 \to \mathbf{List}(\mathbf{Bool}, 5), 3$ then it is not possible to apply f to an argument of type $\mathbf{List}(\mathbf{Bool}, 0)$. The authors suggest that this can be relaxed by conducting separate analysis for each use of f. However, this implies that it is not possible to analyse functions separately from their use, i.e., the analysis is not fully modular. As we will see later, this has been recently solved in [JLHH10].

A more detailed analysis of Hofmann and Jost's type system can be found in Chapter 2 of the PhD thesis by Brian Campbell [Cam08]. But the most in-depth description is to be found in chapters 4 and 5 of the recent PhD thesis by Steffen Jost entitled Automated Amortised Analysis [Jos10].

Hoffmann-Jost based work Hofmann and Jost have considered heap usage but not time or stack usage. Time could, in principle, be treated similarly to heap, by simply recording the number of execution steps instead of the size of a free list. The only difference is the absence of a deallocation mechanism for time costs.

Extending the amortised analysis for stack usage is less straightforward. One technical problem is that a realistic model for stack usage must employ a small-step rather than a big-step semantics as used in [HJ03]. Another concern is that the bounds expressible by the amortised analysis are linear on the size of data structures (the total number of elements). While this is generally a good match for obtaining heap bounds, it will yield coarse stack bounds for a tree search algorithm whose worst-case complexity is linear on the depth of the tree. In [Cam09], Campbell investigates the extension of amortised analysis to stack costs; the definition of potential is modified to keep track of the depth of data structures.

The RAJA type system [HJ06] targets a language with assignment and object-oriented programming features based on FEATHERWEIGHT JAVA extended with updates. Types are annotated with views, where each view maps classes to potential for that particular reference, and assigns a view to each of the fields of the object. The total potential is thus the sum of the potential from these views for each reference over every access path (chain of references) to it. To safely allow updates to references, views are given two kinds of annotations: the first assigns a potential to the variable as usual; the second describes an upper bound on the potential for all aliases of the data structure. Thus on examining a data structure we may use the first amount of potential, but on changing it we supply the second. Conditions are imposed on subtyping so that inheritance and downcasting do not violate the given bounds. An automatic type checking algorithm was later provided by Hofmann and Rodriguez [HR09]. However, the extra complexity would require a more involved inference procedure, especially for the object-oriented features. Thus this system can be used to prove a bound, but not to produce one.

A combination of the amortised analysis for JAVA bytecode and separation logic was proposed by Atkey [Atk10], in order to include the treatment of safety aspects in conjunction with imperative update (see also [AAMS10]).

The Hofmann-Jost analysis has also been adapted to the HUME programming language [Ham02, HM03, HM04a, HFH06a], a functionally-inspired research language for resource-sensitive systems, as part of the EmBounded project. The project's goal is to certify the resource usage of real-time embedded programs written in HUME. In [JLH+09], Loidl and Jost develop an amortised cost based resource analysis for a higher-order, strict functional language. They call their analysis *Carbon Credits* in reference to the attempt of some governments at reducing industrial pollution by issuing tradable carbon credits. A salient feature of this analysis is the possibility to express not only size-dependent but also data-dependent bounds on (generic) resource consumption. Recent improvements and extensions include a call count analysis for higher-order programs, usability aspects like improved presentation of the inferred resource bounds and interactive tuning, and performance improvements [LJ10].

Different variations of the analysis have also been used to infer upper bounds in the heap-space and stack-space consumption and on the worst-case execution time of several embedded systems applications [HFH+06b, HBH+07, JLS+09].

The latest research on the Hofmann-Jost analysis aims at overcoming some of the limitations of the original analysis.

A recent work by Jost, Loidl, Hammond, and Hofmann [JLHH10] extends the analysis to both polymorphic and higher-order types, without requiring source-level transformations. They provide a generic treatment of resource usage through resource tables that can be specialised to different cost metrics and execution models. The ARTHUR analysis (see Chapter 6 of Jost's PhD thesis [Jos10]) also deals with higher-order functions. The major difficulty is that if a function is used several times, then the captured variables from its definition could be used several times (with the same type) and therefore, so will the potential assigned to them. Multiple uses of the same potential could lead to an underestimation of the memory used, thus Jost introduces *additive* (or linear) *pairs*, which allow explicit suspension of evaluation.

In [HH10b], Jan Hoffmann and Martin Hofmann extend the original system to automatically infer *polynomial* resource bounds if the maximal degree is provided. One of the key ideas is to represent a resource (i.e., integer) polynomial of degree k using binomial coefficients as base: $\sum_{i=0..k} a_i \binom{n}{i}$. In a complementary paper [HH10a], they describe an inference algorithm that computes resource-polymorphic types for recursive functions, however, it is not complete with respect to the typing rules. One of the limitations of this approach is that bounds must be sums of univariate polynomials. Multivariate bounds such as $n \cdot m$ must be over-approximated by polynomials like $n^2 + m^2$. A yet to be published work [HAH11] deals with this restriction by developing a multivariate potential-based amortised analysis. They define *multivariate resource polynomials*, that are a generalisation of the resource polynomials used in [HH10b], as a non-negative linear combination of some carefully chosen base of multivariate polynomials. This allows them to obtain bounds for nested inductive data structures.

3 Other Analysis and Techniques

In this section we overview other work on resource analysis that we have organised in six topics. Three of them, automatic complexity analysis, worst case execution time analysis and region analysis can be considered subfields of resource analysis, i.e., they have more concrete goals. The other three topics are dependent types, abstract interpretation and quasi-interpretations. These are fields on their own, whose techniques have been applied to resource analysis. This classification should be taken only as a way to group related research. There are pieces of work that could have been placed in another category because they use more that one of these techniques, for instance, some work on automatic complexity analysis is based on abstract interpretation.

3.1 Automatic Complexity Analysis

Early works in automatic cost analysis follow the methodology for hand analysis of algorithms, e.g., the seminal textbook by Knuth [Knu73]: first derive some recurrence equations expressing the program cost (e.g., number of arithmetic or other primitive operations) in terms of an input metric (e.g., data size) and then solve the recurrences (perhaps using approximation) to obtain a closed equation.

The earliest work following this methodology is Wegbreit's METRIC system [Weg75]. METRIC derives complexity equations for list functions written in a first-order subset of LISP with recursive procedures, but no side effects nor imperative features. The system obtained metrics such as time, length or size as a 4-tuple *(min, max, avg, var)* of lower bound, upper bound, average and standard deviation; the first two are best and worst-case bounds; the last two measures are derived under the assumption of statistical independence of dynamic tests. The performance measures are expressed symbolically as functions of input size or length and the costs of primitive operations.

Le Métayer's ACE system also performs complexity analysis by deriving a recursive step-counting function for each recursive function of the program [Mét88]. It obtains closed-form solutions by a series of meaning-preserving program transformations within a functional calculus (a subset of the FP language). ACE can handle predefined higher-order functions with known costs, however, unlike Wegbreit's approach, only asymptotic[3] worst-case cost measures are considered. Like ACE, Benzinger's work on worst-case complexity analysis supports higher-order functions if the complexity information is provided explicitly [Ben01, Ben04].

Rosendahl describes a semantic-based method for deriving the step-counting version of recursive first-order functions [Ros89]. The main contribution is the use of abstract interpretation to define a time-bound function whose inputs are partial representations of the original program inputs and whose output is an upper-bound on the original program time. No attempt is made to obtain closed cost expressions.

[3] This means that individual primitive operations are not accounted, only the number of recursive calls.

In [Ros06], Ross presents an automatic complexity analysis that recommends the user types and algorithms from the C++ Standard Template Library (STL) that will improve performance. The analysis is based on manipulation and comparison of symbolic complexity expressions, constructed using cost-bound functions and abstract interpretation of program behaviour. Chin et al. [CNPQ08] presented a heap and a stack analysis for a low-level (assembler) language with explicit (de-)allocation. By inferring path-sensitive information and using symbolic evaluation they are able to infer exact stack bounds for all but one example program.

The group of Elvira Albert at the Complutense University of Madrid, have developed the *Cost and Termination Analyser* (COSTA) for JAVA bytecode programs [AAG+07, AGG09]. COSTA tries to infer a symbolic bound of the program's resource consumption, with respect to a given cost model. When performing cost analysis, COSTA expresses upper bounds in the form of a *cost equation system* [AAGP09], which is an extended form of recurrence relations. In order to obtain a closed form for such recurrence relations, COSTA includes a dedicated solver called PUBS [AAGP08]. COSTA as well as the PUBS subsystem provide a web interface, see `https://costa.ls.fi.upm.es/`. A recent publication [ABG+11] investigates the cooperation between COSTA and the KEY [ABHS07] verification tool to automatically produce *verified* resource guarantees.

3.2 Worst Case Execution Time Analysis

Worst-Case Execution Time (WCET) analysis means to compute a safe upper bound to the execution time of a piece of code.

Bonenfant et al. [BFHH07] conducted worst-case execution time (WCET) analysis to obtain bounds on real-time costs for a subset of the abstract machine instructions of HUME [Ham02]. Their approach is to translate the abstract machine instructions into C and use a C compiler to obtain machine code; they then employ AIT, a commercial tool for static WCET analysis of machine code blocks [FHL+01]. Unlike approaches based on experimental tests, AIT uses abstract interpretation to model cache and pipeline states of specific microprocessors and is capable of obtaining guaranteed worst-case time bounds [FMWA99]. Bonenfant et al. applied this tool to derive WCET costs of compiled code for a Renesas M32C/85 micro-controller, compared the results with experimental timings and reported a close match with the analysis bounds.

Lisper [Lis03] describes a technique for fully automatic parametric WCET analysis based on abstract interpretation. Parameters may represent, for instance, values of input parameters to the program, or maximal iteration counts for loops. The technique is capable of handling unstructured code, and it can find upper bounds to loop iteration counts automatically. Shkaravska et al. [SKvE10] presents an interpolation-based method for inferring polynomial bounds of loop iterations in JAVA programs. The essence of the method is to find a "well-chosen"

set of test cases, however, such a set does not always exists. The inferred bounds are provable using an external tool like KeY [ABHS07].

3.3 Region Analysis

Heap *regions* are disjoint parts of the heap that are dynamically allocated and deallocated. A region is empty when created, successively filled with values during its lifetime, and discarded in one go when disposed. Region-based memory management achieves efficiency by bulk allocation and deallocation of objects in memory. However, manual C-like memory management is error-prone and unsafe.

The standard region system of Tofte and Talpin [TT94, TT97, TB98], commonly known as TT, introduces an allocation primitive **letregion** ρ **in** $e[\rho]$, where ρ is a region variable that can used for allocations in e. Operationally, upon entry a new region is allocated and bound to the lexically scoped variable ρ, the expression e is evaluated, the region is deallocated, and the result of $e[\rho]$ is returned to the context of the evaluation (and the variable ρ disappears as its scope is left). The **letregion** construct must be aligned with the program's expression hierarchy, so all region allocations and deallocations follow a stack discipline in unison with the original program's expression structure. TT guarantees that well-typed programs do not access regions after deallocation.

The TT system is efficient for small programs [TT94], but requires a number of extra analyses to give reasonable behaviour for larger programs [BTV96]. The problem is that in practice object lifetimes do not follow a stack discipline. In order to overcome this limitation several mechanisms have been proposed. An extension by Birkedal, Tofte and Vejlstrup [BTV96] allows to reset all the data structures in a region, without deallocating the whole region. In their AFL system, Aiken, Fähndrich, and Levien [AFL95] separate region allocation and deallocation from introduction of region variables. The technique postpones allocation of a region until its first access, and deallocates it just after the last access. This leads to late allocation and early deallocation of regions. In [HMN01], Henglein et al. present a Hoare-style region type system for reasoning about and inferring region-based memory management, using a sublanguage of imperative region commands. The system expresses and performs control-sensitive region management without requiring a stack discipline for allocating and deallocating regions. In all these extensions, optimisation of memory usage requires good knowledge of the internal mechanism.

See [TBEH04] for a comprehensive overview of the systems related to TT, and retrospective view of the authors on the design, implementation and correctness of the ML Kit, a compiler for Standard ML using region-based memory management.

Hughes and Pareto [HP99] extended the type system of Hughes, Pareto and Sabry [HPS96], which was discussed in Section 1, adding effects that approximate stack and heap costs for a strict, first-order functional language called

EMBEDDED ML. The characteristic feature of EMBEDDED ML is that it uses regions for dynamic heap allocation and deallocation.

The combination of sized types and regions allows sizes of regions to be specified at the point of allocation. Thus, the region allocation becomes

$$\mathsf{letregion}\ \rho\#e'\ \mathsf{in}\ e$$

where e' is an expression that specifies the size of region ρ. The type judgements

$$\Gamma \vdash_F e\colon \tau\,!\,\sigma;p;\phi$$

are extended with effects σ, ρ and ϕ: σ is the stack effect, ρ is the put effect and ϕ is the store effect. The stack and store effect are natural numbers and approximate the maximum stack depth and heap allocations during the evaluation of e. The put effect tracks allocations done in regions in the current scope. Γ is a variable type assignment (also known as context) and F is a function type assignment.

Type checking can now ensure at compile-time the absence of space overflow. For example, consider a function that constructs a list of naturals:

$$\begin{aligned} \mathit{nats}\ n\ r = \mathbf{Cons}\ n\ (\mathbf{case}\ n\ \mathbf{of}\ & 0 \to \mathbf{Nil}\ r \\ & \mid m+1 \to \mathit{nats}\ m\ r)\ r \end{aligned}$$

Like in TT, constructors such as **Nil** and **Cons** take an extra argument to specify the region where values are to be allocated. Thus, for example, $\mathit{nats}\ 4\ r$ returns the list $\mathbf{Cons}(4, \mathbf{Cons}(4, \mathbf{Cons}(4, \mathbf{Cons}(4, \mathbf{Nil}))))$ allocated in r. The type checker accepts the typing

$$\mathit{nats}\colon \forall k\ r.\ \mathbf{Nat}_k \times r \to \mathbf{List}_{k+1}(\mathbf{Nat}_k)\ r\ \mathbf{with}\ \sigma = 5k; r_+ = 3k+1$$

which specifies both stack and heap allocation as functions of the size k. The stack effect accounts for 5 stack words at each recursive invocation: one word for each bound variable n, r, m, the intermediate result and the return address. The put effect specifies that the region r can grow by (at most) $3k+1$ heap cells (the constants are derived from the particular operational semantics, which uses 3 words to store a cons cell and one word to store a nil). We now see that the application

$$\mathbf{letregion}\ r\#13\ \mathbf{in}\ \mathit{length}(\mathit{nats}\ 4\ r)$$

is rejected by the type system because the local region r is too small for the computation of *nats*: the size k of the list is 5 instead of 4, because **Nil** must also be counted, so at least $3*5+1 = 16$ heap cells are required. To fix the program it suffices to specify a larger size for r.

The correctness of the type system with respect to an abstract machine based on the SECD [Lan64], can be found in Pareto's PhD thesis [Par00].

One first limitation of this work is inherited from TT: regions follow a stack discipline. Another drawback of Hughes and Pareto's approach is that it requires user annotations of both sizes and costs. While sizes are denotational properties that programmers can reason about in a high-level language, stack and heap costs are dependent on implementation details. Requiring the user to specify

costs in type annotations, even if these are checkable by the compiler, hinders software development. If fully automatic inference is not feasible, it would be preferable to have the user write size annotations and have the system infer costs automatically; this was left as future work in [HP99] and not addressed in [Par00].

SAFE is a first-order strict functional language with regions [PSM06]. It is equipped with a set of compile-time analyses that infers regions where data structures are located and detect when a program with explicit deallocation actions is free of dangling pointers [MPnS08]. Bounds on memory usage are inferred by means of abstract interpretation. The compiler produces JAVA bytecode, hence SAFE programs can be executed in modern mobile devices. Ongoing research aims at inferring a *certified* upper bound on memory consumption [MPnS09, MPnS10, dDMP10].

3.4 Dependent Types

Dependent ML (DML) is a conservative extension of the ML language with dependent types [Xi07, XP99]. DML with integer indices allows expressing properties similar to the sized type systems of Reistad and Gifford [RG94], Hughes et al. [HPS96] or Chin and Khoo [CK01]. There are two important distinctions between the two approaches. Firstly, DML indices are user-definable for each data type whereas the notion of size in the sized type systems above is rigid. And secondly, the DML type checker can *verify* user-annotated size relations but not *infer* them as in [RG94, CK01].

Grobauer [Gro01] presented a method for automatically deriving cost recurrences from first-order DML programs. The main contribution is the use of indices in DML types as data sizes for expressing the recurrences. This allows the user to specify more precise size measures for data, e.g., nested lists or trees. This work focuses on extracting cost recurrences but not on obtaining solutions to the cost equations. Except in very simple cases, obtaining closed form solutions requires human intervention.

Crary and Weirich [CW00] used a system based on proof-carrying code (PCC) [Nec97] to perform verification of resources bounds. This system is based on an intermediate language that allows expressing resource properties in types by exposing a "virtual clock" representing some available resource (e.g., time). Resource properties can then be verified by the type checker. To deal with variable-time procedures, they encode static type-level representations of data using sum and inductive kinds; this simulates type dependency while allowing a simpler theory and type checker. Costs can be expressed as primitive-recursive functions over the static data representations (so that type checking remains decidable). These must be provided by the user: the system allows verifying resource bounds, but makes no attempt to infer them.

Brady and Hammond [BH05] defined a dependently-typed language where the terms depend not only on a size property, but also on a *proof* of that property. They applied this framework to express size relations for higher-order functions

on lists. As it is usual in dependent type systems, types are checked, but not inferred.

Danielsson [Dan08] introduced a library of functions that makes the complexity analysis of a number of purely functional data structures and algorithms almost fully formal. He does this by using a dependent type system to encode information about execution time, and then by combining individual costs into an overall cost using an annotated monad. However, the system requires insightful annotations by the user.

3.5 Abstract Interpretation

While having the attraction of being very general, one major disadvantage of abstract interpretations is that analysis results usually depend on the existence of concrete data values. However, where they can be applied, impressive results can be obtained, even for large commercial applications. For example, AbsInt's aiT tool [FHL+01], and Cousot et al.'s ASTREE system [CCF+05] have both been deployed in the design of the software for the Airbus A380. Typically, such tools are limited to non-recursive programs, and may require significant programmer effort to use effectively.

Huelsbergen, Larus and Aiken [HLA94] have defined an abstract interpretation of a higher-order, strict language for determining computation costs that dependent on the size of data structures. This static analysis is combined with run-time size information to deliver dynamic granularity estimates. Gómez and Liu [GL02] have constructed an abstract interpretation for determining time bounds on higher-order programs. This executes an abstract version of the program that calculates cost parameters, but which otherwise mirrors the normal program execution strategy. Therefore, the cost of this analysis depends directly on the actual values of the input data and the number of iterations that are performed, it does not give a general cost metric for all possible inputs, and fails to terminate when applied to non-terminating programs.

Gulwani, Mehra and Chilimbi's SPEED system [GMC09] uses a symbolic evaluation approach to calculate non-linear complexity bounds for C/C++ procedures using an abstract interpretation-based invariant generation tool. Precise loop bounds are calculated for half of the production loops that have been studied. They target only first-order programs and consider only time bounds. They do, however, consider non-linear bounds and disjunctive combination of cost information.

3.6 Quasi-Interpretations

Basically, a quasi-interpretation a is a function such that (i) a is bounded, (ii) $a(x_1, \ldots, x_n) \geq x_i$ for all $1 \leq i \leq n$ and (iii) a is non-decreasing with respect to each variable. It should be noted that the bound can be a polynomial, in such case it is called a *polynomial quasi-interpretation* [BMM04]. Quasi-interpretations are inspired by *polynomial simplification interpretations* which are one of the

traditional tools used in proving the termination of term rewriting systems, see, e.g., [BN98]. In a nutshell, a quasi-interpretation for a program provides an upper bound on the size of the values computed by the program as a function of the size of its inputs.

Quasi-interpretations were first used in the context of implicit complexity analysis and termination analysis [Ama03, Mar03, BMM04]. Marion et al. have shown that when combined with *recursive path orderings*, polynomial quasi-interpretations entail polynomial complexities in time and/or space. More generally, a quasi-interpretation by itself entails a complexity bound on the computed function, though not a very tight one.

More recently they have also been applied to resource analysis. Amadio et al. [ACGDL04] defined a simple stack machine for a first-order functional language and showed how to perform type, size and termination verifications at the bytecode level. Their main result is a proof that each program with the quasi-interpretation property that terminates has a polynomial stack bound. Their focus is on termination verification, rather than on inference of stack bounds. Furthermore, heap space was not considered. In [Ama05], Amadio uses quasi-interpretation in conjunction with *max-plus algebra*. In a *max-plus polynomial* addition is replaced by *max* and plus takes the role of the multiplicative operator. These polynomials are common in Discrete Event Systems (DES), where the maximum time is taken when waiting for concurrent events, and times are added for events that happen sequentially. Amadio proves that the synthesis problem for max-plus quasi-interpretations is NP-hard in general, but also shows a synthesis algorithm for *multilinear max-plus polynomials* that is NP-complete. Further work on quasi-interpretations applies the technique in a simple language with cooperative multithreading to bound the amount of heap memory used by each thread [AD06]. Quasi-interpretation is combined with standard termination techniques to ensure liveness, and it can also provide what they describe as 'rather rough' stack bound.

Recent research leaded by Marion has generalised quasi-interpretations to sup–interpretations [MP06, MP09], which can be applied to a wider range of programs. However, the analysis is still restricted to monotonic bounds. To the best of our knowledge, *non-monotonic* quasi-interpretations have not been studied for size analysis, but only for proving termination [HM04b].

Chapter 6

Size Analysis of Algebraic Data Types

Revised version of *Size Analysis of Algebraic Data Types.*
In Proceedings of the 9th International Symposium on Trends in Functional Programming (TFP 2008).
Center Parcs "Het Heijderbos", The Netherlands, May 26–28, 2008

Size Analysis of Algebraic Data Types

Alejandro Tamalet[1], Olha Shkaravska[1], Marko van Eekelen[12] *

[1] Institute for Computing and Information Sciences (iCIS),
Radboud University Nijmegen, The Netherlands
[2] School of Computer Science, Open University, The Netherlands

Abstract. We present a size-aware type system for a first-order functional language with algebraic data types, where types are annotated with polynomials over size variables. We define how to generate typing rules for each data type, provided its user defined size function meets certain requirements. As an example, a program for balancing binary trees is type checked. The type system is shown to be sound with respect to the operational semantics in the class of shapely functions. Type checking is shown to be undecidable, however, decidability for a large subset of programs is guaranteed.

1 Introduction

Embedded systems or server applications often have limited resources available. Therefore, it can be important to know in advance how much time or memory a computation is going to take, for instance, to determine how much memory should at least be put in a system to enable all desired operations. This helps to prevent abrupt termination on small devices like mobile phones and Java cards as well as on powerful computers running memory exhaustive computations like GRID applications and model generation. Analysing resource usage is also interesting for optimisations in compilers, in particular optimisations of memory allocation and garbage collection techniques. An accurate estimation of heap usage enables preallocation of larger chunks of memory instead of allocating memory cells separately when needed, leading to a better cache performance. Size verification can be used to avoid memory exhaustion which helps to prevent attacks that exploit it, like some "Denial of Service" attacks. Size-aware type systems can also be used to prove termination of finite computations or progression of infinite ones (see the related work section).

Decisions regarding these (and related) problems should be based on formally verified upper bounds of resource consumption. A detailed analysis of these bounds requires knowledge of the sizes of the data structures used throughout the program (see [SvKvE07a]).

As part of the AHA project, we study in this chapter a *type-and-effect system* [Pie04, NN99] for a strict first-order functional language with algebraic data

* This work is sponsored by the Netherlands Organisation for Scientific Research (NWO) under grant nr. 612.063.511.

types, where types are annotated with size information. We focus on *shapely* function definitions in this language, where shapely means that the size of their output is polynomial with respect to the sizes of its arguments. Formally, if $size_{\tau_i} : \tau_i \to \mathcal{N}$ are the size functions of the types τ_i for $i = 1..k+1$, a function $f : \tau_1 \times \ldots \times \tau_k \to \tau_{k+1}$ is shapely if there exists a polynomial p on k variables such that

$$\forall x_1 : \tau_1, \ldots, x_k : \tau_k \;.\; size_{\tau_{k+1}}(f(x_1, \ldots, x_k)) = p(size_{\tau_1}(x_1), \ldots, size_{\tau_k}(x_k))$$

For instance, if we take for lists their length to be their size, then appending two lists is shapely because the size of the output is the sum of the sizes of the inputs. However, a function that conditionally deletes an element from a list is not shapely because the size of the output can be the same as the size of the input or one less, which can not be expressed with a unique polynomial. The definition can be easily extended to size functions that return tuples of natural numbers.

We have previously shown for a basic language (whose only types are integers and lists) and a simplified size-aware type system, that type checking is undecidable in general, but decidable under a syntactical restriction [SvKvE07a]. Type inference through a combination of dynamic testing and type checking was developed in [vKSvE07]. A demonstrator for type checking and type inference is available at `http://resourceanalysis.cs.ru.nl/`.

In this chapter we extend this analysis to algebraic data types. We show a procedure to generate size-aware typing rules for an algebraic data type, provided its size function has a given form. Furthermore, for any data type we define a *canonical size function* which is used in case no size function is defined by the user. We prove soundness of the type system with respect to its operational semantics, which allows sharing. In the presence of sharing, the size annotations can be interpreted as an upper bound on the amount of memory used to allocate the result. Type checking is shown to be undecidable, however, the syntactic restriction introduced in [SvKvE07a] can be used to guarantee decidability. We also give an example that shows that the type system is incomplete.

This chapter is organised as follows. In Section 2 we define the language and the type system, and we give generic typing rules for user defined size functions. In Section 3 we deal with soundness, decidability and completeness issues. Section 4 discusses a possible extension to the language and future work. Section 5 comments on related work and Section 6 draws conclusions.

2 Size-Aware Type System with Algebraic Data Types

We start this section by introducing the working language and types with size annotations followed by an example with binary trees in 2.2. Subsection 2.3 shows how to obtain typing rules from a size function that meets the requirements stated in 2.1.

2.1 Language and Types

We define a type and effect system in which types are annotated with polynomial size expressions:

$$p ::= c \mid n \mid p + p \mid p - p \mid p * p$$

where c is a rational number and n denotes a size variable that ranges over natural numbers. A zero-order type can be one of the primitive data types (boolean and integers), a type variable or size annotated algebraic data type:

$$\tau ::= \mathtt{Bool} \mid \mathtt{Int} \mid \alpha \mid T^{p_1,\ldots,p_n}(\tau_1, \ldots, \tau_m)$$

An algebraic data type is annotated with a tuple of polynomials. This allows one to measure different aspects of an element of that type, for instance, the number of times each constructor is used. To simplify the presentation we will usually write just $T^{\mathbf{p}}(\overline{\tau})$.

For lists we consider its size as the number of elements it contains. Thus, $\mathsf{List}^n(\alpha)$ is the type of lists of size n and whose elements are of type α and $\mathsf{List}^n(\mathsf{List}^m(\mathtt{Int}))$ is the type of matrices of n by m integers. Note that the types $\mathsf{List}^0(\mathsf{List}^m(\mathtt{Int}))$ are equivalent for all m because their only inhabitant is the empty list. When counting the occurrences of all constructors, we can generalise this to any algebraic data type by regarding as equivalent all elements that have a size of zero in all the non-nullary constructors of the outer type.

For example, if $\mathsf{Tree}^{n,m}(\alpha)$ is the type of binary trees with n leaves and m nodes, elements of type $\mathsf{Tree}^{1,0}(\mathsf{List}^m(\mathtt{Int}))$ are considered equivalent for any m. The canonical value of this class is $\mathsf{Tree}^{1,0}(\mathsf{List}^0(\mathtt{Int}))$. In this work τ denotes in fact the canonical representative $\tau_{\equiv}$.

The sets $FV(\tau)$ and $FSV(\tau)$ of the free type and free size variables of τ, are defined inductively in the obvious way. Note, that $FSV(\mathsf{List}^0(\mathsf{List}^m(\alpha))) = \emptyset$, since this type is equivalent to $\mathsf{List}^0(\mathsf{List}^0(\alpha))$. Let τ° denote a zero-order type whose size annotation contains just constants or size variables. First-order types are assigned to shapely functions over values of τ°-types.

$$\begin{aligned}\tau^f ::= {} & \tau_1^\circ \times \ldots \times \tau_k^\circ \to \tau_{k+1} \\ & \text{such that } FSV(\tau_{k+1}) \subseteq FSV(\tau_1^\circ) \cup \cdots \cup FSV(\tau_k^\circ)\end{aligned}$$

For *total* functions the following condition is necessary: *for all instantiations * of size variables with themselves or zeros,* $FSV(*\tau_{n+1}) \subseteq FSV(*\tau_1^\circ) \cup \cdots \cup FSV(*\tau_n^\circ)$. Consider, e.g., the first-order type $\mathsf{List}^n(\mathsf{List}^m(\alpha)) \to \mathsf{List}^m(\mathsf{List}^n(\alpha))$. When $n = 0$ the input type degenerates to $\mathsf{List}^0(\mathsf{List}^0(\alpha))$, but in the output, the outer list must have size m, which in this case is unknown. Hence, this first-order type may be accepted without the condition on instantiations, only if a function of this type is *undefined* [3] on empty lists. In the previous example the type may correspond to an $n \times m$-matrix transposition function, in which case

[3] We use a nonterminating function to express undefinition.

undefinedness on Nil would be interpreted as the exception "cannot transpose an empty matrix".

We work with a fairly simple first-order language over these types. The following grammar defines the syntax of the language, where b ranges over booleans and $\imath$ over integers, x denotes a program variable of a zero-order type, C stands for a constructor name and f for a function name.

$$
\begin{array}{lll}
d & ::= & \texttt{data}\ T(\overline{\alpha}) = C_1(\overline{\tau}_1(\overline{\alpha})) \mid \ldots \mid C_r(\overline{\tau}_r(\overline{\alpha})) \\
a & ::= & b \mid \imath \mid f(\overline{x}) \mid C(\overline{x}) \\
e & ::= & a \mid \texttt{letfun}\ f(\overline{x}) = e_1\ \texttt{in}\ e_2 \\
 & & \mid \texttt{let}\ x = a\ \texttt{in}\ e \mid \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
 & & \mid \texttt{match}\ x\ \texttt{with} \mid C_1(\overline{x}_1) \Rightarrow e_{C_1} \mid \ldots \mid C_r(\overline{x}_r) \Rightarrow e_{C_r} \\
pr & ::= & d^*\ e
\end{array}
$$

In the data type definition we have abused the notation: only type constructors may have type variables as parameters. This *let-normal form* imposed to allow maximum simplicity from a proof-theoretic viewpoint while covering all the essentials of a functional programming language. It can be achieved by introducing additional let expressions, e.g. writing $\texttt{let}\ y = g(x)\ \texttt{in}\ f(y)$ instead of the more succinct program term $f(g(x))$. Types appearing on the right hand side of the definition of a data type must not have free size variables. We prohibit head-nested let-expressions and restrict subexpressions in function calls to variables to make type checking straightforward. Program expressions of a general form can be equivalently transformed into expressions of this form. It is useful to think of this as an intermediate language. We also assume that the language has the typical basic operations on integers and booleans, but their study is omitted since they do not involve size annotations.

In order to add size annotations to an algebraic data type, it must be decided what to measure. Because of polymorphism, one can measure only the outer structure, e.g., since the size of $\mathsf{List}(\alpha)$ must be defined for any α, the size of a $\mathsf{List}(\mathsf{Tree}(\texttt{Int}))$ will be just the length of the list. But, because the size is part of the type, all the elements of the list must have the same size, which allows the user to compute the total size once the sizes of the trees are known.

One usually needs to count the number of times each constructor is used to build an element. A size function for

$$\texttt{data}\ \mathsf{TreeAB}(\alpha, \beta) = \mathsf{Empty} \mid \mathsf{Leaf}(\alpha) \mid \mathsf{Node}(\beta, \mathsf{TreeAB}(\alpha, \beta), \mathsf{TreeAB}(\alpha, \beta))$$

should return the number of empties, leaves and nodes. Any size function for these trees that returns a single natural number is losing information and the user will not be able to calculate the total size once α and β are known. One may not want to count the number of times some constructor is used because it can be deduced from the others or it is constant, e.g., any finite list has always one nil constructor cell. Ignoring some constructors can also make a function definition shapely as in the case of a function that can return trees of type TreeAB with different number of empties and leaves, but always the same number of nodes.

If all the constructors cells are counted, such a function is not shapely, however, if only nodes are counted, it is shapely.

We require a size function for $T(\overline{\alpha})$ to be total and have the form

$$size_T(C_i(x_{i1}, \ldots, x_{ik_i})) = \mathsf{c}_i + \sum_{j=1}^{k_i} \gamma(x_{ij})$$

where $x_{ij}\colon T_{ij}$, c_i is a non-negative integer or a tuple of non-negative integers and

$$\gamma(x_{ij}) = \begin{cases} size_T(x_{ij}) & \text{if } T_{ij}(\overline{\alpha}) = T(\overline{\alpha}) \\ 0 & \text{otherwise} \end{cases}$$

Henceforth, we will assume that every size function satisfies these requirements. The motivation for this is twofold. On one hand linearity is needed for decidability (see 3.2) and on the other hand, requiring the recursive calls of the size function to be applied to (some of) the arguments of the constructors, allows us to relate their sizes with the annotations of the respective types in the context (see 2.3).

A *canonical size function* for $T(\overline{\alpha})$ is a size function where each c_i is $\mathbf{1}_i^r$, the tuple of arity r (the number of constructors of T) with all zeros except for a 1 on the i-th position. It is always possible to obtain a canonical size function for a given algebraic data type, and there is only one way to construct it, thus it is unique for that type. When no size function for a type is provided by the user, its canonical size function is used. We write s_T for the canonical size function of $T(\overline{\alpha})$. For instance, s_{List} is a function that takes a list l and returns $(1, \mathit{length}(l))$, since it is defined as:

$$\begin{aligned} s_{\mathsf{List}}(\mathsf{Nil}) &= (1, 0) \\ s_{\mathsf{List}}(\mathsf{Cons}(hd, tl)) &= (0, 1) + s_{\mathsf{List}}(tl) \end{aligned}$$

We say that a size function for T is *sensible* if it returns the exact amount of occurrences of each constructor of T in its argument. Recall that an inductive type is an initial algebra of the endofunctor corresponding to its constructors [BW90]. Because we do not allow free size variables in the definitions of data types, we can always "flatten" any algebraic data type of our language, and obtain an isomorphic *polynomial inductive type*. It is not difficult to prove [TSvE08] that the canonical size function of an polynomial inductive type is sensible.

The syntax distinguishes between zero-order let-binding of variables (`let`) and first-order letfun-binding of functions (`letfun`). In a function body, the only free program variables that may occur are its parameters: $FV(e_1) \subseteq \{x_1, .., x_n\}$. The operational semantics is standard, therefore the definition is postponed until it is used to prove soundness (Section 3).

A context Γ is a finite mapping from zero-order variables to zero-order types. A signature Σ is a finite function from function names to first-order types. A typing judgement is a relation of the form $D;\ \Gamma \vdash_\Sigma e\colon \tau$, where D is a set of *Diophantine* equations (i.e., with integer solutions) that constrains the possible

values of the size variables, where $vars(D) \subseteq FSV(\tau_1^\circ \times \ldots \times \tau_k^\circ)$, and Σ contains a type assumption for the function that is going to be type checked along with the signatures of the functions used in its definition. When D is empty we will write $\Gamma \vdash_\Sigma e\colon \tau$. The entailment $D \vdash \mathsf{p} = \mathsf{p}'$ means that $\mathsf{p} = \mathsf{p}'$ is derivable from the equations in D, while $D \vdash \tau = \tau'$ means that τ and τ' have the same underlying type and equality of their size annotations is derivable. We write the union of the constraints c_1 and c_2 as c_1, c_2, and we write Γ_1, Γ_2 to denote the union of the contexts Γ_1 and Γ_2, provided $dom(\Gamma_1) \cap dom(\Gamma_1) = \emptyset$.

The typing rules for the language, excluding the ones for data types, are shown in Figure 1.

$$\frac{}{D;\ \Gamma \vdash_\Sigma b\colon \texttt{Bool}}\ \textsc{BConst} \qquad \frac{}{D;\ \Gamma \vdash_\Sigma i\colon \texttt{Int}}\ \textsc{IConst}$$

$$\frac{D \vdash \tau = \tau'}{D;\ \Gamma, x\colon \tau \vdash_\Sigma x\colon \tau'}\ \textsc{Var}$$

$$\frac{\Gamma(x) = \texttt{Bool} \qquad D;\ \Gamma \vdash_\Sigma e_t\colon \tau \qquad D;\ \Gamma \vdash_\Sigma e_f\colon \tau}{D;\ \Gamma \vdash_\Sigma \texttt{if } x \texttt{ then } e_t \texttt{ else } e_f\colon \tau}\ \textsc{If}$$

$$\frac{x \notin dom(\Gamma) \qquad D;\ \Gamma \vdash_\Sigma e_1\colon \tau_x \qquad D;\ \Gamma, x\colon \tau_x \vdash_\Sigma e_2\colon \tau}{D;\ \Gamma \vdash_\Sigma \texttt{let } x = e_1 \texttt{ in } e_2\colon \tau}\ \textsc{Let}$$

$$\frac{\begin{array}{c}\Sigma(f) = \tau_1^\circ \times \cdots \times \tau_k^\circ \to \tau_{k+1} \\ x_1\colon \tau_1^\circ, \ldots, x_k\colon \tau_k^\circ \vdash_\Sigma e_1\colon \tau_{k+1} \qquad D;\ \Gamma \vdash_\Sigma e_2\colon \tau'\end{array}}{D;\ \Gamma \vdash_\Sigma \texttt{letfun } f(x_1, \ldots, x_k) = e_1 \texttt{ in } e_2\colon \tau'}\ \textsc{LetFun}$$

$$\frac{\begin{array}{c}\Sigma(f) = \tau_1^\circ \times \ldots \times \tau_k^\circ \to \tau_{k+1} \\ D \vdash \tau = \tau_{k+1}[\tau_1^\circ := \tau_1', \ldots, \tau_k^\circ := \tau_k'] \qquad D \vdash C\end{array}}{D;\ \Gamma, x_1\colon \tau_1', \ldots, x_k\colon \tau_k' \vdash_\Sigma f(x_1, \ldots, x_k)\colon \tau}\ \textsc{FunApp}$$

Fig. 1. Typing rules excluding the ones for data types.

The FunApp rule needs some comments on its notation: $\tau[\tau_1^\circ := \tau_1' \ldots \tau_k^\circ := \tau_k']$ is the simultaneous substitution in τ of τ_i° by τ_i' for $i = 1..k$. This is done as follows:

1. Check that the underlying type (i.e., the type without the size annotations) of τ_i° and τ_i' are the same, except for the type variables.
2. Check that every type variable in each τ_i° is substituted by the same zero-order type by each τ_i', and substitute them in τ.
3. Substitute in τ the size variable of τ_i° by the corresponding size expression in τ_i'. It may happen that the same size variable appears in different types τ_i° and τ_j°, and that it is substituted by size expressions p_i and p_j, respectively. In such a case, replace the size variable by p_i and add the equation $\mathsf{p}_i = \mathsf{p}_j$ to C (which is initially empty).

As example, consider the last step in type checking *append* (see [SvKvE07b]).

$$\frac{\begin{array}{c}\Sigma(\mathsf{Cons}) = \alpha' \times \mathsf{List}^{n'}(\alpha') \to \mathsf{List}^{n'+1}(\alpha') \\ D \vdash \mathsf{List}^{n+m}(\alpha) = \mathsf{List}^{n'+1}(\alpha')[\alpha'/\alpha, \mathsf{List}^{n'}(\alpha')/\mathsf{List}^{(n-1)+m}(\alpha)]\end{array}}{h\colon \alpha,\ z\colon \mathsf{List}^{(n-1)+m}(\alpha) \vdash_\Sigma \mathsf{Cons}(h,\ z)\colon \mathsf{List}^{n+m}(\alpha)}\ \textsc{FunApp}$$

To do the substitutions we follow the 3 steps described before. As a first step we check the consistency of the underlying types of the *actuals* and the *formals*. Since we omit type variables, there is nothing to check in α'/α; and it is obvious that $\mathsf{List}^{n'}$ and $\mathsf{List}^{(n-1)+m}$ have the same underlying type, List. Then we check that each type variable is instantiated to the same value, which is true since in both cases α' is instantiated to α. We replace α' by α in $\mathsf{List}^{n'+1}(\alpha')$. Finally we replace n' by $(n-1)+m$ to get the entailment $\vdash \mathsf{List}^{n+m}(\alpha) = \mathsf{List}^{(n-1)+m+1}(\alpha)$. Since there is only one substitution of n', the set of equations C is empty.

The implicit side-conditions on FunApp, i.e., the checks of consistency of the underlying types made in the first two steps, will be omitted in the following examples because they are part of conventional type checking. We will concentrate on checking the entailments about size expressions.

The set C is used e.g., when type checking a function to do matrix multiplications: $\mathsf{List}^{n}(\mathsf{List}^{k}(\mathtt{Int})) \times \mathsf{List}^{k}(\mathsf{List}^{m}(\mathtt{Int})) \to \mathsf{List}^{n}(\mathsf{List}^{m}(\mathtt{Int}))$. If such a function is instantiated with lists of type $\mathsf{List}^{p_1}(\mathsf{List}^{p_2}(\mathtt{Int}))$ and $\mathsf{List}^{q_1}(\mathsf{List}^{q_2}(\mathtt{Int}))$, we add the condition $p_2 = q_1$ to C.

In [SvKvE07a] we defined a type system for a similar language, whose only data type were lists and integers. The typing rules for calculating the size of lists were "hardcoded" in the type system by the typing rules in Figure 2. The main contribution of this work is to extend the type system to cope with other algebraic data types.

$$\frac{D \vdash p = 0}{D;\ \Gamma \vdash_\Sigma \mathsf{Nil}\colon \mathsf{List}^{p}(\tau)}\ \textsc{Nil}$$

$$\frac{D \vdash p = q + 1}{D;\ \Gamma,\ hd\colon \tau,\ tl\colon \mathsf{List}^{q}(\tau) \vdash_\Sigma \mathsf{Cons}(hd,\ tl)\colon \mathsf{List}^{p}(\tau)}\ \textsc{Cons}$$

$$\frac{\begin{array}{c}D,\ p = 0;\ \Gamma,\ x\colon \mathsf{List}^{p}(\tau) \vdash_\Sigma e_{\mathsf{Nil}}\colon \tau' \qquad hd,\ tl \notin dom(\Gamma) \\ D;\ \Gamma,\ x\colon \mathsf{List}^{p}(\tau),\ hd\colon \tau,\ tl\colon \mathsf{List}^{p-1}(\tau) \vdash_\Sigma e_{\mathsf{Cons}}\colon \tau'\end{array}}{D;\ \Gamma,\ x\colon \mathsf{List}^{p}(\tau) \vdash_\Sigma \begin{array}{l}\mathtt{match}\ x\ \mathtt{with} \mid \mathsf{Nil} \Rightarrow e_{\mathsf{Nil}} \\ \qquad\qquad\quad \mid \mathsf{Cons}(hd,\ tl) \Rightarrow e_{\mathsf{Cons}}\end{array} : \tau'}\ \textsc{Match}$$

Fig. 2. Typing rules for lists.

2.2 Example: Binary Trees

Consider the following definition of binary trees:

$$\mathtt{data}\ \mathsf{Tree}(\alpha) = \mathsf{Empty} \mid \mathsf{Node}(\alpha,\ \mathsf{Tree}(\alpha),\ \mathsf{Tree}(\alpha))$$

The canonical size function for Tree is:

$$\begin{aligned} s_{\mathsf{Tree}}(\mathsf{Empty}) &= (1,\ 0) \\ s_{\mathsf{Tree}}(\mathsf{Node}(v,\ l,\ r)) &= (0,\ 1) + s_{\mathsf{Tree}}(l) + s_{\mathsf{Tree}}(r) \end{aligned}$$

Conforming to s_{Tree}, an annotated binary tree has the form $\mathsf{Tree}^{e,n}(\alpha)$, where e is the number of Empty constructors (the leaves of the tree) and n is the number of nodes. We want to obtain typing rules for binary trees that will enable us to statically check the values of e and n when the binary tree is the result of a shapely function. We need one rule per constructor and one rule for pattern matching a binary tree. An empty tree has one leaf and no node, thus:

$$\frac{D \vdash (e,\ n) = (1,\ 0)}{D;\ \Gamma \vdash_\Sigma \mathsf{Empty}\colon \mathsf{Tree}^{e,n}(\tau)}\ \textsc{Empty}$$

From s_{Tree} we obtain that in a non-empty tree, the number of leaves is equal to the sum of the number of leaves in each subtree and that the number of nodes is one more than the sum of the number of nodes in each subtree. We use variables for the sizes of the subtrees and we relate them accordingly in the premise:

$$\frac{D \vdash (e,\ n) = (0,\ 1) + (e_1,\ n_1) + (e_2,\ n_2)}{D;\ \Gamma,\ v\colon \tau,\ l\colon \mathsf{Tree}^{e_1,n_1}(\tau),\ r\colon \mathsf{Tree}^{e_2,n_2}(\tau) \vdash_\Sigma \mathsf{Node}(v,\ l,\ r)\colon \mathsf{Tree}^{e,n}(\tau)}\ \textsc{Node}$$

Similarly, in the typing rule for pattern matching a binary tree, we introduce fresh variables in the typing context of the premises for the unknown quantities and we add their relationship to the set of conditions.

$$\frac{\begin{array}{c} D,\ (e,\ n) = (1,\ 0);\ \Gamma,\ t\colon \mathsf{Tree}^{e,n}(\tau) \vdash_\Sigma e_{\mathsf{Empty}}\colon \tau' \\ D,\ (e,\ n) = (0,\ 1) + (e_1,\ n_1) + (e_2,\ n_2);\ \Gamma, \\ t\colon \mathsf{Tree}^{e,n}(\tau),\ v\colon \tau,\ l\colon \mathsf{Tree}^{e_1,n_1}(\tau),\ r\colon \mathsf{Tree}^{e_2,n_2}(\tau) \vdash_\Sigma e_{\mathsf{Node}}\colon \tau' \\ e_1,\ e_2,\ n_1\ n_2 \notin \mathit{vars}(D) \qquad v,\ l,\ r \notin \mathit{dom}(\Gamma) \end{array}}{D;\ \Gamma,\ t\colon \mathsf{Tree}^{e,n}(\tau) \vdash_\Sigma \begin{array}{l} \mathtt{match}\ t\ \mathtt{with} \mid \mathsf{Empty} \Rightarrow e_{\mathsf{Empty}} \\ \qquad\qquad\quad \mid \mathsf{Node}(v,\ l,\ r) \Rightarrow e_{\mathsf{Node}} \end{array} : \tau'}\ \textsc{MTree}$$

To see how these rules work in practice, we apply them to a function to balance a (not necessarily ordered) binary tree. To simplify the example we add syntactic sugar to avoid `let` constructs. It is not our intention to explain the balancing algorithm, but just to show that there are many interesting functions that can be written in our language. We begin with a function for right-rotation

of nodes. We use undefined to indicate a non-terminating expression with the required type.

$$r\_rot(v,l,r) : \alpha \times \mathsf{Tree}^{e_1,n_1}(\alpha) \times \mathsf{Tree}^{e_2,n_2}(\alpha) \to \mathsf{Tree}^{e_1+e_2,n_1+n_2+1}(\alpha) =$$
$$\texttt{match}\ l\ \texttt{with} \mid \mathsf{Empty} \Rightarrow \mathsf{undefined}$$
$$\mid \mathsf{Node}(v_1,\ l_1,\ r_1) \Rightarrow \mathsf{Node}(v_1,\ l_1,\ \mathsf{Node}(v,\ r_1,\ r))$$

By applying the rule MTree we get two branches. The branch for the Empty case is undefined and thus we do not need to type check it. Instead of writing undefined we could have written a call to the function with the same arguments, in this case $l\_rot(v,\ l,\ r)$. In that case that branch would be still undefined and the type checker would trivially succeed. For the Node branch we have:

$$\frac{\begin{array}{l}(e_1,n_1) = \\ (e_{11}+e_{12},\ n_{11}+n_{12}+1)\end{array} \vdash \begin{array}{l}(e_1+e_2,\ n_1+n_2+1) = \\ (e_{11}+e_{12}+e_2,\ n_{11}+(n_{12}+n_2+1)+1)\end{array}}{\begin{array}{l}(e_1,\ n_1) = \\ (e_{11}+e_{12},\ n_{11}+n_{12}+1); \\ v,\ v_1 \colon \alpha,\ l \colon \mathsf{Tree}^{e_1,n_1}(\alpha), \\ l_1 \colon \mathsf{Tree}^{e_{11},n_{11}}(\alpha), \\ r_1 \colon \mathsf{Tree}^{e_{12},n_{12}}(\alpha)\end{array} \vdash_\Sigma \begin{array}{l}\mathsf{Node}(v_1, l_1, \\ \mathsf{Node}(v,\ r_1,\ r))\end{array} : \mathsf{Tree}^{e_1+e_2,n_1+n_2+1}(\alpha)}\ \textsc{Node}$$

$$\frac{\begin{array}{l}v \colon \alpha,\ l \colon \mathsf{Tree}^{e_1,n_1}(\alpha), \\ r \colon \mathsf{Tree}^{e_2,n_2}(\alpha)\end{array} \vdash_\Sigma \texttt{match}\ l \ldots : \mathsf{Tree}^{e_1+e_2,n_1+n_2+1}(\alpha)}{\text{(Node branch)}}\ \textsc{MTree}$$

Similarly, we can type check the left-right rotation function. For simplicity we write it in a Haskell-like style of pattern matching.

$$lr\_rot \colon \alpha \times \mathsf{Tree}^{e_1,n_1}(\alpha) \times \mathsf{Tree}^{e_2,n_2}(\alpha) \to \mathsf{Tree}^{e_1+e_2,n_1+n_2+1}(\alpha)$$
$$lr\_rot(v,\ \mathsf{Node}(v_1,\ l_1,\ \mathsf{Node}(v_{12},\ l_{12},\ r_{12})),\ r) =$$
$$\mathsf{Node}(v_{12},\ \mathsf{Node}(v_1,\ l_1,\ l_{12}),\ \mathsf{Node}(v,\ r_{12},\ r))$$

Now we define the left balance function, which is easily type checked since both branches have the same type. The definitions of *balance* and RightWeight are omitted because they are not needed for our analysis.

$$l\_bal(v,\ l,\ r) \colon \alpha \times \mathsf{Tree}^{e_1,n_1}(\alpha) \times \mathsf{Tree}^{e_2,n_2}(\alpha) \to \mathsf{Tree}^{e_1+e_2,n_1+n_2+1}(\alpha) =$$
$$\texttt{if}\ \mathit{balance}(l) == \mathsf{RightWeight}$$
$$\texttt{then}\ lr\_rot(v,\ l,\ r)$$
$$\texttt{else}\ r\_rot(v,\ l,\ r)$$

Then we type check a function that inserts an element into a balanced binary tree:

$$
\begin{array}{l}
\mathit{insert}(a,\, t)\colon \alpha \times \mathsf{Tree}^{e,n}(\alpha) \to \mathsf{Tree}^{e+1,n+1}(\alpha) = \\
\quad \texttt{match}\ t\ \texttt{with} \mid \mathsf{Empty} \Rightarrow \mathsf{Node}(a,\, \mathsf{Empty},\, \mathsf{Empty}) \\
\qquad\qquad\qquad \mid \mathsf{Node}(v,\, l,\, r) \Rightarrow \mathsf{let}\ l_2 = \mathit{insert}(a,\, l) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ \texttt{if}\ \mathit{height}(l_2) == \mathit{height}(r) + 2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{then}\ l\_\mathit{bal}(v,\, l_2,\, r) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{else}\ \mathsf{Node}(v,\, l_2,\, r)
\end{array}
$$

Applying MTree we get two branches. For the Empty branch we get the entailment $(e,\, n) = (1,\, 0) \vdash (e+1,\, n+1) = (1+1,\, 0+0+1)$ and for the Node branch we have the judgement:

$$
\begin{array}{l}
(e,\, n) = (e_1 + e_2,\, n_1 + n_2 + 1);\ t\colon \mathsf{Tree}^{e,n}(\alpha), \\
v\colon \alpha,\, l\colon \mathsf{Tree}^{e_1,n_1}(\alpha),\, r\colon \mathsf{Tree}^{e_2,n_2}(\alpha)
\end{array}
\vdash_{\Sigma} \mathsf{let}\ l_2 = \ldots : \mathsf{Tree}^{e+1,n+1}(\alpha)
$$

Using Let we get $l_2\colon \mathsf{Tree}^{e_1+1,n_1+1}(\alpha)$. Both branches of the `if` have the same type, so we only need to check the entailment it generates:

$$(e,\, n) = (e_1 + e_2,\, n_1 + n_2 + 1) \vdash (e+1,\, n+1) = ((e_1+1)+e_2,\, (n_1+1)+n_2+1)$$

Then we define a function to build a balanced tree from a list:

$$
\begin{array}{l}
\mathit{build\_bal\_tree}(xs) : \mathsf{List}^{n}(\alpha) \to \mathsf{Tree}^{n+1,n}(\alpha) = \\
\quad \texttt{match}\ xs\ \texttt{with} \mid \mathsf{Nil} \Rightarrow \mathsf{Empty} \\
\qquad\qquad\qquad\quad \mid \mathsf{Cons}(hd,\, tl) \Rightarrow \mathit{insert}(hd,\, \mathit{build\_bal\_tree}(tl))
\end{array}
$$

From the Nil branch we get the condition $n = 0 \vdash (n+1,\, n) = (1,\, 0)$, which is trivially true and for the Cons branch we have:

$$
\frac{\vdash (n+1,\, n) = ((n-1)+1+1,\, (n-1)+1)}{hd\colon \alpha,\, tl\colon \mathsf{List}^{n-1}(\alpha) \vdash_{\Sigma} \mathit{insert}(hd,\, \mathit{build\_bal\_tree}(tl)) : \mathsf{Tree}^{n+1,n}(\alpha)}\ \textsc{FunApp}
$$

Finally, we define and type check a function that balances a binary tree:

$$\mathit{balance\_tree}(t)\colon \mathsf{Tree}^{e,n}(\alpha) \to \mathsf{Tree}^{n+1,n}(\alpha) = \mathit{build\_bal\_tree}(\mathit{flatten}(t))$$

where *flatten* is a function with type $\mathsf{Tree}^{e,n}(\alpha) \to \mathsf{List}^{n}(\alpha)$ that returns a list with the elements of a binary tree. By applying the typing rule for function application twice, we get the trivial entailment $\vdash (n+1,\, n) = (n+1,\, n)$. When the tree is flattened, we loose the information about e, thus e does not appear in the resulting type of *balance_tree*.

For Tree it does not make sense to count both constructors because if e and n are the number of Empty and Node constructors in any Tree, respectively, it

holds that $e = n + 1$. However, in general there is no such relationship. As an example where counting different constructors is relevant, consider binary trees defined as:

$$\mathtt{data}\ \mathsf{Tree}(\alpha) = \mathsf{Empty} \mid \mathsf{Leaf}(\alpha) \mid \mathsf{Node}(\mathsf{Tree}(\alpha),\ \mathsf{Tree}(\alpha))$$

Suppose that we annotate Tree with e, l and n representing the number of empties, leaves and nodes, respectively. A function that replaces all empties with a leaf has type $\alpha \times \mathsf{Tree}^{e,l,n} \to \mathsf{Tree}^{0,e+l,n}$.

2.3 Typing Rules for Algebraic Data Types

Below, we give a procedure for obtaining typing rules for an arbitrary algebraic data type. Let $T(\overline{\alpha})$ be an algebraic data type defined as

$$\mathtt{data}\ T(\overline{\alpha}) = C_1(\overline{\tau}_1(\overline{\alpha})) \mid \ldots \mid C_r(\overline{\tau}_r(\overline{\alpha}))$$

and let $size_T$ be the size function of $T(\overline{\alpha})$. For each constructor C_i we add a typing rule of the form

$$\frac{D \vdash \mathsf{p} = \mathsf{c}_i + \sum_{j=1}^{k_i} \mathsf{p}_{ij}}{D;\ \Gamma,\ x_{ij} \colon \gamma'_{ij}(T(\overline{\tau}))\ \text{ for } j = 1..k_i \vdash_\Sigma C_i(x_{i1}, \ldots, x_{ik_i}) \colon T^{\mathsf{p}}(\overline{\tau})}\ \mathrm{C}_i \quad \text{for } i = 1..r$$

where c_i and the x_{ij} are taken from the definition of $size_T$, and γ'_{ij} is defined as

$$\gamma'_{ij}(T(\overline{\tau})) = \begin{cases} T^{\mathsf{p}_{ij}}(\overline{\tau}) & \text{if } \tau_{ij}(\overline{\tau}) = T(\overline{\tau}) \\ \tau_{ij}(\overline{\tau}) & \text{otherwise} \end{cases}$$

The idea is that if the type of x_{ij} is $T(\overline{\tau})$, the one we are defining the typing rules for, then it must have a tuple of fresh size variables that we call p_{ij}, otherwise its type is just $\tau_{ij}(\overline{\tau})$. For example, in the NODE rule x_{22} is l, i.e. the second argument of the second constructor, and because it is of type Tree, $\gamma'_{22}(\mathsf{Tree}(\alpha)) = \mathsf{Tree}^{\mathsf{p}_{22}}(\alpha)$. In the previous example we chose $\mathsf{p}_{22} = (e_1, n_1)$.

There is a clear correspondence between γ and γ'.

We also add a typing rule for pattern matching an element of type $T(\overline{\alpha})$:

$$\frac{\begin{array}{l} D,\ \mathsf{p} = \mathsf{c}_i + \sum_{j=1}^{k_i} \mathsf{n}_{ij};\ \Gamma,\ x \colon T^{\mathsf{p}}(\overline{\tau}), \\ \quad x_{ij} \colon \gamma'_{ij}(T(\overline{\tau}))\ \text{ for } j = 1..k_i \end{array} \vdash_\Sigma e_i \colon \tau' \quad \text{for } i = 1..r \qquad \mathsf{n}_{ij} \notin vars(D),\ x_{ij} \notin dom(\Gamma)\ \text{ for } i = 1..r, j = 1..k_i}{\begin{array}{ll} & \mathtt{match}\ x\ \mathtt{with} \mid C_1(x_{11}, \ldots, x_{1k_1}) \Rightarrow e_1 \\ D;\ \Gamma,\ x \colon T^{\mathsf{p}}(\overline{\tau}) \vdash_\Sigma & \quad\vdots \qquad\qquad\qquad : \tau' \\ & \mid C_r(x_{r1}, \ldots, x_{rk_r}) \Rightarrow e_r \end{array}}\ \textsc{MatchT}$$

Each of the size variables of n_{ij} and the formal parameters of the constructors are assumed to be fresh. Notice that there is one premise per constructor. When $\gamma_{ij}(T(\overline{\tau}))$ is $\tau_{ij}(\overline{\tau})$ we regard n_{ij} as $\mathsf{0}$, that is, we omit that variable from the sum.

Instead of generating one typing rule for each constructor, it is possible to derive a size-annotated type for each of them like in [HPS96], add these types to the set of signatures Σ and then use the function application rule. This approach results in a type system with fewer rules, however, for presentation purposes, we have chosen to generate typing rules for them because it makes the role clearer that the set of constraints D plays. A typing rule for pattern matching each algebraic data type is still needed.

3 Soundness, Decidability and Completeness

This section is devoted to soundness and completeness of the type system and decidability of type checking, extending previous results on these topics to a language with algebraic data types.

3.1 Soundness

Set-theoretic heap-aware semantics of a ground algebraic data type (i.e., a type where all size and type variables are instantiated) is an obvious extension of the semantics of lists that can be found, for instance, in [SvKvE07a]. Intuitively, an instance of a ground type is presented in a heap as a directed tree-like structure, that may overlap with other structures. The only restriction is that it must be acyclic. Note that cyclic structures may be studied as, for instance, in the paper of Hofmann and Jost [HJ06] about heap-space analysis for a subset of JAVA.

Since our type system is not linear, that is, a program variable may be used more than once, a data structure in a heap may consist of overlapping substructures. This is the case, for instance, for a heap representation of $\mathsf{Node}(1, t, t)$, where t is a non-empty tree. In general, in a calculation of the *size* of a structure, a node is counted as many times as it is referenced. Hence, a sensible size function gives an upper bound for the actual amount of constructor cells allocated by the structure. If there is no internal sharing, the sensible size function is equal to the amount of cells actually allocated.

A location is the address of some constructor-cell of a ground type. A *program value* is either an integer or boolean constant, or a location. A *heap* is a finite partial mapping from locations and fields to program values, and an *object heap* is a finite partial map from locations to *Constructor*, the set of (the names of) constructors. Below, we assume that for any heap h, there is an object heap oh such that $dom(h) = dom(oh)$.

Let τ be a type defined by a set of constructors C_i, where $1 \leq i \leq r$. With a constructor C_i of arity $k_i > 0$, we associate a collection of field names $C_i\_field_j$, where $1 \leq j \leq k_i$. Let *Field* be the set of all field names in a given program. To avoid technical overhead with semantics of null-ary constructors, we do not

consider a null-address `NULL` as a program value. The problem is that a type may have more than one null-ary constructor. If one had a `NULL`-address, then, to avoid ambiguity, one of the null-ary constructors would have been associated with `NULL`, whereas the others would have been placed in some locations. This would have made the proofs non-uniform. We also assume that any null-ary constructor is placed in a location with 1 empty integer field. With a 0-arity constructor C_i we associate the field name $C_i\_field_1$. The reason to introduce a "fake" field for null-ary constructors is to make the proofs more uniform. Formally:

$$\begin{array}{llll} Val\ v & ::= & \imath \mid b \mid \ell & \ell \in Loc \qquad \imath \in \texttt{Int} \qquad b \in \texttt{Bool} \\ Heap\ h & : & Loc \rightharpoonup Field \rightharpoonup Val & ObjHeap\ oh \ : \ Loc \rightharpoonup Constructor \end{array}$$

We will write $h[\ell.field := v]$ for the heap equal to h everywhere but in ℓ, which at the field of ℓ named *field* gets the value v.

The semantics w of a program value v is a set-theoretic interpretation with respect to a specific heap h, its object heap oh and a ground type $\tau^\bullet$, via a five-place relation $v \models^{h;oh}_{\tau^\bullet} w$. Integer and boolean constants interpret themselves, and locations are interpreted as non-cyclic structures:

$$\imath \models^{h;\,oh}_{\texttt{Int}} \imath$$

$$b \models^{h;oh}_{\texttt{Bool}} b$$

$$\ell \models^{h;\,oh}_{T^{\mathsf{c}}(\tau^\bullet)} C \qquad \text{if } C \text{ is a null-ary constructor of } T,\ \ell \in dom(h),\ oh(\ell) = C \text{ and the constant vector } \mathsf{c} \text{ is the size of } C$$

$$\ell \models^{h;\,oh}_{\tau^\bullet} C(w_1, \ldots,\ w_k) \qquad \text{if } \ell \in dom(h),\ oh(\ell) = C,\ C\colon \tau_1^\bullet \times \ldots \times \tau_k^\bullet \to \tau^\bullet \text{ (i.e. it is a ground instance)}, \ \tau^\bullet = T^{\mathsf{n}^0}(\overline{\tau^\bullet}') \text{ for some } \overline{\tau^\bullet}',\ \mathsf{n}^0 = size_T(C(w_1, \ldots, w_k)), \text{ and for all } 1 \le j \le k\colon \ h.\ell.C\_field_j \models^{h|_{dom(h)\setminus\{\ell\}};\ oh|_{dom(oh)\setminus\{\ell\}}}_{\tau_j^\bullet} w_j$$

where $h|_{dom(h)\setminus\{\ell\}}$ denotes the heap equal to h everywhere except for ℓ, where it is undefined.

When a function body is evaluated, a frame store maintains the mapping from program variables to values. At the beginning it contains only the actual function parameters, thus preventing access beyond the caller's frame. Formally, a frame store is a finite partial map from variables to values: $Store\ s\colon ExpVar \rightharpoonup Val$.

An operational semantics judgement $s;\ h;\ oh,\ \mathcal{C} \vdash e \rightsquigarrow v;\ h';\ oh'$ informally means that at a store s, a heap h, its object heap oh and with the set $\mathcal{C}$ of function closures (bodies), the evaluation of an expression e terminates with value v in the heap h' and object heap oh'.

Using heaps and frame stores, and maintaining a mapping $\mathcal{C}$ from function names to bodies for the functions definitions encountered, the operational semantics of expressions is shown in Figure 3.

Let a valuation $\epsilon : SizeVar \to \mathcal{Z}$ map size variables to concrete sizes (integer numbers) and an instantiation $\eta : TypeVar \to \tau^\bullet$ map type variables to ground types. Applied to a type, context, or size expression, valuation and instantiation

$$\frac{c \in \texttt{Int}}{s;\ h;\ oh,\ \mathcal{C} \vdash c \rightsquigarrow c;\ h;\ oh}\ \text{OSICONST} \qquad \frac{}{s;\ h;\ oh,\ \mathcal{C} \vdash x \rightsquigarrow s(x);\ h;\ oh}\ \text{OSVAR}$$

$$\frac{C_i \text{ is a 0-ary constructor} \qquad \ell \notin dom(h)}{s;\ h;\ oh,\ \mathcal{C} \vdash C_i \rightsquigarrow \ell;\ h[\ell.C_i\_field_1 := i];\ oh[\ell := C_i]}\ \text{OSCONS}-0$$

$$\frac{s(x_1) = v_1, \ldots, s(x_k) = v_k \qquad \ell \notin dom(h)}{s;\ h;\ oh,\ \mathcal{C} \vdash C(x_1, \ldots, x_k) \rightsquigarrow \ell;\ h[\ell.C\_field_1 := v_1, \ldots, \ell.C\_field_k := v_k];\ oh[\ell := C]}\ \text{OSCONS}$$

$$\frac{s(x) \neq 0 \qquad s;\ h;\ oh,\ \mathcal{C} \vdash e_1 \rightsquigarrow v;\ h';\ oh'}{s;\ h;\ oh,\ \mathcal{C} \vdash \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow v;\ h';\ oh'}\ \text{OSIFTRUE}$$

$$\frac{s(x) = 0 \qquad s;\ h;\ oh,\ \mathcal{C} \vdash e_2 \rightsquigarrow v;\ h';\ oh'}{s;\ h;\ oh,\ \mathcal{C} \vdash \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow v;\ h';\ oh'}\ \text{OSIFFALSE}$$

$$\frac{s;\ h;\ oh,\ \mathcal{C} \vdash e_1 \rightsquigarrow v_1;\ h_1;\ oh_1 \qquad s[x := v_1];\ h_1;\ oh_1,\ \mathcal{C} \vdash e_2 \rightsquigarrow v;\ h';\ oh'}{s;\ h;\ oh,\ \mathcal{C} \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rightsquigarrow v;\ h';\ oh'}\ \text{OSLET}$$

$$\frac{\begin{array}{c}C_i \text{ is a 0-ary constructor in the collection } C_{i'},\ 1 \leq i' \leq r \\ oh(s(x)) = C_i \qquad s;\ h;\ oh,\ \mathcal{C} \vdash e_i \rightsquigarrow v;\ h';\ oh'\end{array}}{\begin{array}{l}s;\ h;\ oh,\ \mathcal{C} \vdash \texttt{match}\ x\ \texttt{with} \mid C_1(x_{11}, \ldots, x_{1k_1}) \Rightarrow e_1 \rightsquigarrow v;\ h';\ oh' \\ \qquad\qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad\qquad \mid C_r(x_{r1}, \ldots, x_{rk_r}) \Rightarrow e_r\end{array}}\ \text{OSMATCH}-C_i-0$$

$$\frac{\begin{array}{c}oh(s(x)) = C_i \qquad h.s(x).C_i\_field_1 = v_1,\ \ldots,h.s(x).C_i\_field_{k_i} = v_{k_i} \\ s[x_1 := v_1, \ldots, x_{k_i} := v_{k_i}];\ h;\ oh,\ \mathcal{C} \vdash e_i \rightsquigarrow v;\ h';\ oh'\end{array}}{\begin{array}{l}s;\ h;\ oh,\ \mathcal{C} \vdash \texttt{match}\ x\ \texttt{with} \mid C_1(x_{11}, \ldots, x_{1k_1}) \Rightarrow e_1 \rightsquigarrow v;\ h';\ oh' \\ \qquad\qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad\qquad \mid C_r(x_{r1}, \ldots, x_{rk_r}) \Rightarrow e_r\end{array}}\ \text{OSMATCH} - C_i$$

$$\frac{s;\ h;\ oh,\ \mathcal{C}[f := ((x_1, \ldots, x_n) \times e_1)] \vdash e_2 \rightsquigarrow v;\ h';\ oh'}{s;\ h;\ oh,\ \mathcal{C} \vdash \mathsf{letfun}\ f((x_1, \ldots, x_n)) = e_1\ \mathsf{in}\ e_2 \rightsquigarrow v;\ h';\ oh'}\ \text{OSLETFUN}$$

$$\frac{\begin{array}{c}\mathcal{C}(f) = (y_1, \ldots, y_n) \times e_f \qquad FV(e_f) \subseteq \overline{y} \\ [y_1 := s(x_1), \ldots, y_n := s(x_n)];\ h;\ oh,\ \mathcal{C} \vdash e_f \rightsquigarrow v;\ h';\ oh'\end{array}}{s;\ h;\ oh,\ \mathcal{C} \vdash f(x_1, \ldots, x_n) \rightsquigarrow v;\ h';\ oh'}\ \text{OSFUNAPP}$$

Fig. 3. Operational-semantics rules.

map all variables occurring in it to their valuation and instantiation images: $\epsilon(p[+,-,*]p) = \epsilon(p)[+,-,*]\epsilon(p)$ and $\eta(\epsilon(T^{\mathsf{p}}(\tau))) = T^{\epsilon(\mathsf{p})}(\eta(\tau))$.

The soundness statement is defined by means of the following two predicates. One indicates whether a program value is meaningful with respect to a certain heap and ground type. The other does the same for sets of values and types, taken from a frame store and ground context:

$$\begin{aligned} &\mathit{Valid}_{\mathsf{val}}(v,\, \tau^{\bullet},\, h;\, oh) &&= \exists\, w.\ v \models^{h,\, oh}_{\tau^{\bullet}} w \\ &\mathit{Valid}_{\mathsf{store}}(\mathit{vars},\, \Gamma,\, s,\, h; oh) &&= \forall\, x \in \mathit{vars}.\ \mathit{Valid}_{\mathsf{val}}(s(x),\, \Gamma(x),\, h,\, oh) \end{aligned}$$

Now, stating soundness of the type system is straightforward:

Theorem 1. *Let s; h; oh, $[\,] \vdash e \rightsquigarrow v$; h'; oh'. Then for any context Γ, signature Σ, and type τ, such that* True; $\Gamma \vdash_{\Sigma} e\colon \tau$ *is derivable in the type system, and any size valuation ϵ and type instantiation η, it holds that if the store is meaningful w.r.t. the context $\eta(\epsilon(\Gamma))$ then the output value is meaningful w.r.t. the type $\eta(\epsilon(\tau))$:*

$$\forall\, \eta, \epsilon.\ \mathit{Valid}_{\mathsf{store}}(FV(e),\, \eta(\epsilon(\Gamma)),\, s,\, h,\, oh) \implies \mathit{Valid}_{\mathsf{val}}(v,\, \eta(\epsilon(\tau)),\, h',\, oh')$$

The proof is done by induction on the operational-semantics tree. It can be found in Section 1 of the appendix.

3.2 Decidability

Type checking using the type system studied in this work seems to be straightforward because for every syntactic construction of the language there is only one applicable typing rule. The procedure ultimately reduces to proving equations involving rational polynomials.

Lemma 1. *The type checking problem D; $\Gamma \vdash_{\Sigma} e\colon \tau$ can be reduced to checking a finite number of entailments of the form $D' \vdash \mathsf{p} = \mathsf{q}$, where the variables in D',* p *and* q *are either free size variables of Γ or size variables introduced during the type checking procedure.*

Proof. By induction on the structure of the language. □

But consider the following expression, where $f_i\colon \mathsf{List}^{n_1}(\alpha_1) \times \ldots \times \mathsf{List}^{n_k}(\alpha_k) \to \mathsf{List}^{p_i(n_1,\ldots,n_k)}(\alpha)$ for $i = 0, 1, 2$, (assuming we count only the number of elements).

$$\begin{aligned} &\texttt{let}\ x = f_0(x_1, \ldots, x_k)\ \texttt{in match}\ x\ \texttt{with} \mid \mathsf{Nil} \Rightarrow f_1(x_1, \ldots, x_k) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{Cons}(hd,\, tl) \Rightarrow f_2(x_1, \ldots, x_k) \end{aligned}$$

When checking whether this expression has type $\mathsf{List}^{n_1}(\alpha_1) \times \ldots \times \mathsf{List}^{n_k}(\alpha_k) \to \mathsf{List}^{p(n_1,\ldots,n_k)}(\alpha)$, in the Nil branch we will get the entailment

$$p_0(n_1, \ldots, n_k) = 0 \vdash p(n_1, \ldots, n_k) = p_1(n_1, \ldots, n_k)$$

To validate this entailment we must know whether p_0 has roots or not (that is, whether the Nil branch can be entered at all). This is necessary to type check, for instance, a function definition where $p \equiv 0$ and $p_1 \equiv 1$. In [SvKvE07a] it is shown that for *any* given polynomial q, it is possible to construct a function f_0 whose result has as size annotation the polynomial $p_0 = q^2$, whose roots are exactly the ones of q. Hence, type checking reduces to solving Hilbert's tenth problem and thus it is undecidable.

The source of the problem in the previous example was that the pattern match was done over a variable bound by a `let`. We can avoid these cases with a syntactical restriction that we call *no-let-before-match*: given a function body, allow pattern matching only on the function parameters or variables bound by other pattern matchings. Even with this restriction, one can write all shapely primitive recursive functions for our data types because they induce a (polynomial) functor. For instance, the operator for primitive recursion on lists is defined as follows:

$$\begin{aligned} f(x, \overline{y}) = \texttt{match}\ x\ \texttt{with} &\mid \mathsf{Nil} \Rightarrow g(\overline{y}) \\ &\mid \mathsf{Cons}(hd,\ tl) \Rightarrow h(hd,\ tl,\ \overline{y},\ f(tl,\ \overline{y})) \end{aligned}$$

where g and h are functions already defined, and $\overline{y}$ is a sequence of parameters. It is obvious that f satisfies the syntactic restriction. However, we want to emphasise that this condition is sufficient, but not necessary for decidability.

This condition can be enforced by a more restrictive grammar where the `let`-construct in e is replaced by $\texttt{let}\ x = b\ \mathsf{in}\ e_{nomatch}$, where

$$\begin{aligned} e_{nomatch} := {} & b \\ \mid {} & \texttt{let}\ y = b\ \mathsf{in}\ e_{nomatch} \\ \mid {} & \texttt{if}\ y\ \texttt{then}\ e_{nomatch}\ \texttt{else}\ e_{nomatch} \\ \mid {} & \texttt{letfun}\ f(x_1, \ldots, x_n) = e\ \texttt{in}\ e_{nomatch} \end{aligned}$$

For this reason we call the syntactic condition *no-let-before-match*.

We say that a set of constraints is *linear* if each constraint is of the form $\mathsf{n} = \mathsf{c} + \sum_{i=1}^{k} a_i \cdot \mathsf{n}_i$, where the components of n and n'_i are either constants or size variables and c is a tuple of constants.

Lemma 2. *If D is linear then type checking $D; \Gamma \vdash_\Sigma e : \tau$, reduces to checking a set of entailments of the form $D' \vdash \mathsf{p} = \mathsf{q}$, where D' is linear.*

Proof. By Lemma 1 we know that the type checking problem terminates with a set of entailments of the form $D' \vdash \mathsf{p} = \mathsf{q}$. We prove the linearity of D' by induction on the structure of the language. Except for the `match` case, the result follows from the induction hypothesis, since $D' = D$.

Assume that e is an expression of the form $\texttt{match}\ x\ \texttt{with}\ \ldots$ and that x has type T^{p} in the context. The MATCH generates new judgements where for each constructor C_i, $D' = D$, $\mathsf{p} = \mathsf{c}_i + \sum_{i=1}^{k_i} a_{ij} \cdot \mathsf{n}_{ij}$. Thus, it only remains to prove that p is indeed a constant or a size variable (not any polynomial).

If x is free in e or it is the parameter of a function, then it must have a τ°-type on the context. From the definition of τ°-types, `p` is a tuple with size variables or constants. Otherwise, due to the syntactic restriction, e can not be a subexpression of a `let`-body and thus x must be bound by another `match`. Hence, there is a variable y and a superexpression e' of e, such that

$$\begin{aligned} e' = \texttt{match } y \texttt{ with } & \mid C_1(x_{11}, \ldots, x_{1k_1}) \Rightarrow e_1 \\ & \vdots \\ & \mid C_l(x_{l1}, \ldots, x, \ldots, x_{lk_l}) \Rightarrow e_l \\ & \vdots \\ & \mid C_r(x_{r1}, \ldots, x_{rk_r}) \Rightarrow e_r \end{aligned}$$

being e a subexpression of e_l. But then the match rule applied to the judgement containing e' added a fresh size variable as the size annotation of x, and it is obvious that no rule can change that. Thus, when we get to type checking e, `p` is a size variable. □

The MATCH rule has in its premises judgements where the size of the variable being matched is expressed in D as a linear combination of new variables. Any of these variables can in turn be further subdivided, creating a *tree-decomposition* of a size variable as shown in Figure 4.

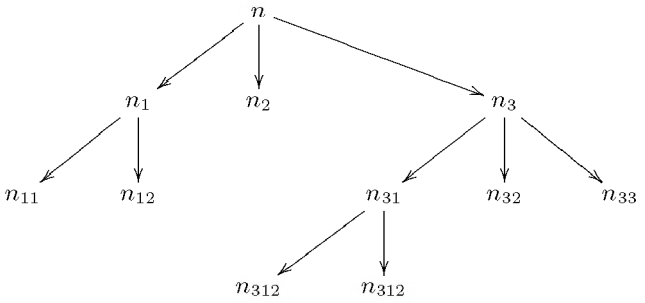


Fig. 4. Example of a tree-decomposition of a size variable. The edges mean that the father is a linear combination of the children.

The *no-let-before-match* restriction ensures that only size variables, and not polynomials on them, are linearly decomposed. The previous lemma tells us that after applying the typing rules, the set of constraints in the entailments left to check, contain only tree-decomposition of (some of) the free size variables of Γ. With this lemma it is easy to prove decidability of type checking the restricted language.

Theorem 2. *Type checking an expression that conforms to the* no-let-before-match *restriction is decidable.*

Proof. Let e be an expression that satisfies the syntactic condition, which we want to type check. At the beginning of the type checking procedure the set of constraints is empty and thus it is trivially linear. By Lemma 2, type checking e reduces to checking a set of entailments of the form $D' \vdash \mathsf{p} = \mathsf{q}$, where D' is linear. Then we replace the variables in p and q using the equations in D', following a breadth-first order in the tree-decomposition of each size variable, until we get to an equality of two expression that depend only on the leaves of these trees. But since each variable is substituted by a linear combination, the two expression are polynomials on the leaves of the tree-decompositions. Finally, asserting the value of the equality reduces to comparing the coefficients of the polynomials. □

3.3 Completeness

The type system is not complete: there are shapely functions for which shapeliness cannot be proved by means of the typing rules and arithmetic. This comes as no surprise if we consider that the type system subsumes Peano arithmetic. Another reason for incompleteness is that the typing rule for `if` does not keep any size information obtainable from the condition. Consider, for instance, the following schema of expressions, where $f(x)$ is a list of integers:

$$\texttt{let}\, z = f(x)\, \texttt{in if}\ \mathit{length}(z) == 0\ \texttt{then}\ z\ \texttt{else}\ \mathsf{Nil}$$

These expressions have type $\mathsf{List}^0(\texttt{Int})$, however, the type checker fails to acknowledge it.

4 Discussion and Future Work

In this section we discuss a variation of our type system, a possible extension and future work.

One possible extension to the language and the type system is to add *size-parametric data types*, i.e., types that are parametrised by a tuple of size variables that can be used as size annotations in the definition of the type.

An m-ary tree is a tree where each node has m subtrees. We say that a tree of height h is *h-full* if all the leaves are at height h. When the height is not relevant, we say that it is *full*. We can define m-ary full trees as a size-parametric data type.

$$\texttt{spdata}\ \mathsf{MFullTree}_m(\alpha) = \mathsf{Empty} \mid \mathsf{Node}(\alpha, \mathsf{List}^m(\mathsf{MFullTree}_m(\alpha)))$$

It is clear that this defines m-ary trees. They are also full because the subtrees at the same level must all have the same size. Assuming that we are counting the occurrences of each constructor[4], it is not hard to come up with typing rules for MFullTree.

[4] Since the number of nodes in an m-ary full tree depends on its height, any function that re-shapes one of these trees will have size annotations involving logarithms. Therefore, for this data structure it would be better to define its size as its height.

$$\frac{D \vdash (e,\, n) = (1,\, 0)}{D;\, \Gamma \vdash_\Sigma \mathsf{Empty}\colon \mathsf{MFullTree}_m^{e,n}(\tau)}\ \textsc{Empty}$$

$$\frac{D \vdash (e,\, n) = (0,\, 1) + m * (e',\, n')}{D;\, \Gamma,\, v\colon \tau,\, ts\colon \mathsf{List}^m(\mathsf{MFullTree}_m^{e',n'}(\tau)) \vdash_\Sigma \mathsf{Node}(v,\, ts)\colon \mathsf{MFullTree}_m^{e,n}(\tau)}\ \textsc{Node}$$

$$\frac{\begin{array}{c} D,\, (e,\, n) = (1,\, 0);\, \Gamma,\, t\colon \mathsf{MFullTree}_m^{e,n}(\tau) \vdash_\Sigma e_{\mathsf{Empty}}\colon \tau' \\ D,\, (e,\, n) = (0,1) + m * (e',\, n');\, \Gamma,\, t\colon \mathsf{MFullTree}_m^{e,n}(\tau),\, v\colon \tau,\, ts\colon \mathsf{List}^m(\mathsf{MFullTree}_m^{e',n'}(\tau)) \vdash_\Sigma e_{\mathsf{Node}}\colon \tau' \\ e',\, n' \notin vars(D) \qquad v,\, ts \notin dom(\Gamma) \end{array}}{D;\, \Gamma,\, t\colon \mathsf{MFullTree}_m^{e,n}(\tau) \vdash_\Sigma \begin{array}{l} \mathtt{match}\ t\ \mathtt{with} \mid \mathsf{Empty} \Rightarrow e_{\mathsf{Empty}} \\ \qquad\qquad\quad \mid \mathsf{Node}(v,\, ts) \Rightarrow e_{\mathsf{Node}} \end{array} : \tau'}\ \textsc{MMFTree}$$

A size function for MFullTree counting both constructors is defined below:

$$\begin{array}{lll} size & : \mathsf{MFullTree}_m(\alpha) \to \mathcal{N} \times \mathcal{N} & \\ size(\mathsf{Empty}) & = (1,\, 0) & \\ size(\mathsf{Node}(v,\, ts)) & = (0,\, 1) + m * \mathtt{match}\ ts\ \mathtt{with} & \mid \mathsf{Nil} \Rightarrow (0,\, 0) \\ & & \mid \mathsf{Cons}(hd,\, tl) \Rightarrow size(hd) \end{array}$$

But there is no direct relationship between this size function and the previous typing rules. The size function used in the typing rules is simpler because the size of the subtrees can be obtained from the typing context. In order to restore the relationship, we can add a parameter to the size function representing the size of the subtrees.

$$\begin{array}{ll} size & : \mathsf{MFullTree}_m(\alpha) \times (\mathcal{N} \times \mathcal{N}) \to \mathcal{N} \times \mathcal{N} \\ size(\mathsf{Empty},\, (e',\, n')) & = (1,\, 0) \\ size(\mathsf{Node}(v,\, ts),\, (e',\, n')) & = (0,\, 1) + m * (e',\, n') \end{array}$$

This procedure can be applied to other data types, but the generalisation is not elegant. Although this extension would add some expressiveness to the language, its usefulness is not clear since the added types are quite restricted (note, e.g., that a node at the bottom of an m-ary full tree has list of m Empty subtrees).

We believe that our results on *type inference* for size-annotated lists (see [vKSvE07]) can be easily extrapolated to ordinary inductive types. Furthermore, we want to extend our current implementation to deal with data types both in the canonical way and by allowing user defined size functions. Our long term goals are to study type systems annotated with upper and lower bound sizes and to investigate shapeliness in the context of imperative languages.

5 Related Work

Amortised heap space analysis has been developed for linear bounds by Hofmann and Jost [HJ03]. Precise knowledge of sizes is required to extend this approach to

non-linear bounds [vESvK+07]. Brian Campbell [Cam08] extended this approach to infer bounds on *stack* space usage.

A type system based on amortised complexity analysis of heap-space requirements for a Java-like language with explicit deallocation is studied by Hofmann and Jost in [HJ06]. Their approach is to use *views* to assign a potential to each possible path expression, avoiding explicit manipulation of size expressions. They cope with inheritance and aliasing and circular data structures, but they do not treat type inference and the potential is an over-approximation. Another type system for an object-oriented language with a deallocation primitive is presented by Chin et al. [CNQM05], which incorporates an alias control via usage aspects.

Some interesting initial work on inferring size relations within the output of XML transformations has been done by Su and Wassermann [SW04]. Although this work does not yield input-output dependencies, it is able to infer size relations within the output type, for instance if two branches have the same number of elements. Herrmann and Lengauer presented a size analysis for functional programs over nested lists [HL01]. However, they do not solve recurrence equations in their size expressions, as this is not important for their goal of program parallelisation.

Other work on size analysis has been restricted to monotonic dependencies. In *type-based termination* analysis e.g., it is enough to assure that the size (more precisely, an upper bound of it) of a data structure decreases in a recursive call. Research by Pareto has yielded an algorithm to automatically check sized types where linear size expression are upper bounds [Par98]. In the thesis of Abel [Abe06] ordinals above ω are considered as well (they are used, e.g., for types like streams). The language of (ordinal) size expressions for zero-order types in this work is rather simple: it consists of ordinal variables, ordinal successor, and an ordinal limit (see also [Abe09]). This is enough for termination analysis, however for heap consumption analysis more sophisticated size expressions are needed. Construction of non-linear upper bounds using a traditional type system approach has been presented by Hammond and Vasconcellos [VH04], but this work leaves recurrence equations unsolved and it is limited to monotonic dependencies. The work on quasi-interpretations by Bonfante et al. [BMM04] and Amadio [Ama05] also requires monotonic dependencies.

The EmBounded project aims to identify and certify resource-bounded code in Hume, a domain-specific high-level programming language for real-time embedded systems. In his thesis, Pedro Vasconcelos [Vas08] uses abstract interpretation to automatically infer linear approximations of the sizes of recursive data types and the stack and heap of recursive functions written in a subset of Hume.

Exact input-output size dependencies have also been explored by Jay and Sekanina [JS97]. In this work, a shapely program is translated into a program involving sizes. Thus, the relation between sizes is given as a program. However, deriving an arithmetic function from it is beyond the scope of this chapter. In a closely related work [JC94], Jay and Cockett study shapely types, i.e., those whose data and data can be separated in a categorical setting. A notable

difference is that we do not consider a type shapely per se, instead its size function determines whether it is shapely.

An application of exact size information is *load distribution* for parallel computation. For instance, size information helps to distribute a storage effectively and to safely store vector fragments [CBF91].

6 Conclusions

We studied an type and effect type system with size annotations for a first-order functional language. We provided generic typing rules for algebraic data types based on user defined size functions and we proved soundness of the type system with respect to the operational semantics. Our choice to allow (not necessarily monotonic) polynomials as size annotations brings undecidability to type checking, however, it was shown that for a wide range of programs, decidability of type checking functions with algebraic data types can be ensured. Our experience is that in practice, the entailments obtained while type checking are easily solvable.

Although the practical applicability of this work is limited, it explores the current limits of the field. It is also an step towards our goal of providing a practical resource analysis. Its main limitation is that it requires size dependencies to be exact. We are working on an extension of the type system that allows to express lower and upper bounds by specifying a family of indexed polynomials extended with the max operator.

Chapter 7

Collected Size Semantics for Functional Programs over Lists

Revised version of *Collected Size Semantics for Functional Programs over Lists.*
In Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages (IFL 2008). University of Hertfordshire, United Kingdom, 10–12 September, 2008.

Collected Size Semantics for Functional Programs over Lists

Olha Shkaravska[1], Marko van Eekelen[12], and Alejandro Tamalet[1] *

[1] Institute for Computing and Information Sciences (iCIS),
Radboud University Nijmegen, The Netherlands
[2] School of Computer Science, Open University, The Netherlands

Abstract. This work introduces collected size semantics of strict functional programs over lists. The collected size semantics of a function definition is a multivalued size function that collects the dependencies between every possible output size and the corresponding input sizes. Such functions annotate types and are defined by conditional rewriting rules generated during type inference. We focus on the connection between the rewriting rules and lower and upper bounds on the multivalued size functions, when the bounds are given by piecewise polynomials.
Using collected size semantics we are able to infer non-monotonic and non-linear lower and upper polynomial bounds for many functional programs. As a feasibility study, we use the procedure to infer lower and upper polynomial size-bounds on typical functions of a list library.

1 Introduction

Estimating heap consumption is an active research area as it becomes more and more of an issue in many applications, e.g. distributed computing and programming for small devices like smart cards, mobile phones or embedded systems.

This chapter explores typing support for checking output-on-input size dependencies for function definitions (functions for short) in a strict functional language. Knowing lower and upper bounds of these dependencies one can apply *amortisation* [Oka98] to check and infer tight non-linear bounds on heap consumption [vESvK+07]. One of the novel ideas of this chapter is to generate during type inference a set of *multivalued size functions* defined by conditional multiple-choice rewriting rules, which are used to annotate types. Since one is mostly interested in lower and upper bounds for size functions, we establish connections between the rewriting rules and size bounds. We focus on piecewise polynomial bounds that can be described by a finite number of polynomials. Given a set of conditional multiple-choice rewriting rules, we show how to infer lower and upper bounds that define an *indexed family of polynomials*. Such a family fully covers the size function induced by the rewriting rules in the sense

* This work is part of the AHA project which is sponsored by the Netherlands Organisation for Scientific Research (NWO) under grant nr. 612.063.511.

that for each input there is a polynomial in the family that describes the size of the output. In this chapter we refer to such coverage as an approximation.

We work with strict functions over *matrix-like* lists of lists, i.e., every nested list must have the same length. (We think it is possible to relax this restriction by allowing lambda-abstractions in size annotations, see [SvET08b].) We can deal with higher-order functions only when the size of the output depends just on zero-order arguments.

This work continues a series of papers where we have studied output-on-input polynomial size dependencies, in which the polynomials are not necessary monotonic. In [SvKvE07a] we designed a type system where each type is annotated with a single polynomial size expression. It allows to type function definitions where the size of the output depends on the sizes of the inputs, but not on their values. For instance, $\mathsf{append}\colon \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha)$, whereas delete (which deletes from a list the first occurrence of an element if it exists) does not have a type in that system since it may or may not delete an element. We also developed a test-based annotation inference procedure for that system in [vKSvE07].

Exploring Size Functions

To get a global idea of the results of this chapter, consider again the function delete. Its collected size semantics can be expressed by the multivalued function $f_{\mathsf{delete}}(n) = \{n, \max_0(n-1)\}$, where n denotes the length of the input list and $\max_0(n) = \max(0, n)$. Our type system allows to express and infer such multivalued size functions in the form of non-deterministic rewriting rules. For instance,

$$\begin{aligned} &\vdash f_{\mathsf{delete}}(0) \to 0 \\ n \geq 1 &\vdash f_{\mathsf{delete}}(n) \to n-1 \mid 1 + f_{\mathsf{delete}}(n-1) \end{aligned}$$

However, users prefer to deal with size functions in *closed form*, i.e. without recursion, like $f(n) = \{n, \max_0(n-1)\}$, or with their lower and upper bounds. The problem of obtaining closed forms for rewriting rules does not have a general solution. We study how to approximate a closed-form solution with an indexed family of piecewise polynomials, if such an approximation exists. For f_{delete} we can infer the family $\{\max_0(n-i)\}_{0 \leq i \leq 1}$, which precisely describes it.

Let $\overline{n}$ denote a vector of variables of the form $(n_1, \ldots, n_k)$. The inference procedure is based on the well-known fact that a multivariate polynomial p of degree d is defined by a finite number of points of the form $\{(\overline{n}_i, p(\overline{n}_i))\}_{i=0}^{m}$ where $m = \binom{d+k}{k} - 1$, that determine a system of linear equations w.r.t. the polynomial coefficients. It takes the following parameters: the *degree* d of polynomial lower and upper bounds of the size function, an *initial point* $\overline{n}^0$ and a *step* to obtain the next point. Using these parameters, the procedure generates the points lying on the bounds. For instance, for delete, we choose degree 1, initial point $n^0 = 1$ and step 1. Then the procedure generates the test points $\overline{n}^i$. In the example it generates $n^0 = 1$, $n^1 = 2$. Next, the rewriting rules are used to calculate the sets $f(\overline{n}^i)$. For delete we obtain $f(n^0) = \{0, 1\}$ and $f(n^1) = \{1, 2\}$. The

procedure picks up the minimal and maximal values from each of the set $f(\overline{n}^i)$ and computes the coefficients of the lower and upper polynomial bounds as the solutions of two corresponding linear systems. In the example, the lower bound $p_{\min}(n) = n - 1$ is computed from the nodes $\{(1,0), (2,1)\}$, and the upper bound $p_{\max}(n) = n$ from $\{(1,1), (2,2)\}$. The obtained bounds $p_{\min}$ and $p_{\max}$ define an indexed family of polynomials, which may be presented, for instance, as $\{p_{\min}(\overline{n}) + i\}_{i=0}^{\delta(\overline{n})}$, where $\delta(\overline{n}) = p_{\max}(\overline{n}) - p_{\min}(\overline{n})$. The procedure depends on user-defined parameters (an initial point, a step, a degree). The consequences of a bad choice of parameters is that the bound will not be tight or they may even be incorrect. Checking if a given indexed family of polynomials covers a given size function is shown to be similar to type checking types annotated with indexed families of polynomials [SvET08b].

The rest of this chapter is organised as follows. In Section 2 we define the programming language and in Section 3 its size-aware type system. Section 3 also defines the semantics of program values w.r.t. zero-order types and the operational semantics of the language. We give an inference procedure for families of polynomials that approximate multivalued size functions and discuss examples in Section 4. In Section 4.2 we discuss the feasibility of applying the analysis to a typical list library. Related work is discussed in Section 5. Section 6 draws conclusions and gives directions to future work. The technical report [SvET08a] gives more examples of checking and inference in detail.

2 Language

The type system is designed for a strict functional language over integers and (polymorphic) lists. Algebraic data types could be added as we did in the previous chapter. Language expressions are defined by the following grammar:

$$
\begin{array}{lll}
\textit{Basic}\ b & ::= & c \mid \mathsf{unop\ x} \mid \mathsf{x\ binop\ y} \mid \mathsf{Nil} \mid \mathsf{Cons(z,\ l)} \mid \mathsf{f}(\mathsf{g}_1, \ldots, \mathsf{g}_l, \mathsf{z}_1, \ldots, \mathsf{z}_k) \\
\textit{Expr}\ e & ::= & b \mid \mathsf{if\ x\ then}\ e_1\ \mathsf{else}\ e_2 \\
 & & \mid \mathsf{let\ z} = b\ \mathsf{in}\ e_1 \\
 & & \mid \mathsf{match\ l\ with} \mid \mathsf{Nil} \Rightarrow e_1 \\
 & & \qquad\qquad \mid \mathsf{Cons}(\mathsf{z}_{\mathsf{hd}},\ \mathsf{l}_{\mathsf{tl}}) \Rightarrow e_2 \\
 & & \mid \mathsf{letfun\ f}(\mathsf{g}_1, \ldots, \mathsf{g}_l, \mathsf{z}_1, \ldots, \mathsf{z}_k) = e_1\ \mathsf{in}\ e_2
\end{array}
$$

where c ranges over integer and boolean constants False and True, x and y denote program variables of integer and boolean types, l ranges over lists, z denotes a program variable of zero-order type, i.e., an integer a boolean or a size annotated list, g ranges over higher-order program variables, i.e., functions, unop is a unary operation, either $-$ or $\neg$, binop is one of the integer or boolean binary operations, and f denotes a function name.

The syntax distinguishes between zero-order let-binding of variables and higher-order letfun-binding of functions. In a function body, the only free program variables are its parameters. We prohibit head-nested let-expressions and we require a *let-normal form* which can be achieved by simple transformations, as explained in Chapter 6.

3 Type System

We consider a type system constituted from zero-order and higher-order types for which the size of the output does not depend on the size of the higher-order inputs. Zero-order types are assigned to program values, which are integers, booleans and finite lists.

$$\textit{Types} \quad \tau ::= \mathsf{Int} \mid \mathsf{Bool} \mid \alpha \mid \mathsf{L}_s(\tau)$$

where α is a type variable and s is a size annotation. Size annotations represent multivalued functions $s\colon \mathcal{R}^k \to 2^{\mathcal{R}}$ that represent lengths of finite lists. $\mathcal{R}$ can be any numerical ring; its choice influences decidability of type checking and the set of well-typed programs [SvET08b]. They are generated by the following grammar:

$$\textit{Sizes} \quad s ::= c \mid n \mid f(s_1, \ldots, s_n) \mid -s_1 \mid s_1 \ \mathbf{binop}\ s_2$$

where c is a constant in $\mathcal{R}$, n is a size variable, f is a function symbol, and binop is addition, subtraction or multiplication in $\mathcal{R}$. In our type system the meaning of a function symbol f is given by conditional rewriting rules. For example, consider a function insert that inserts an element z into a list l if and only if there is no element in l related to z by g.

```
insert(g, z, l) =
  match l with | Nil ⇒ let l' = Nil in Cons(z, l')
               | Cons(hd, tl) ⇒ if g(z, hd) then l else
                                let l'' = insert(g, z, tl) in Cons(hd, l'')
```

The corresponding size rewriting system is

$$\begin{aligned} &\vdash f_{\mathsf{insert}}(0) \to 1 \\ n \geq 1 &\vdash f_{\mathsf{insert}}(n) \to n \mid 1 + f_{\mathsf{insert}}(n-1) \end{aligned}$$

The type of insert is $(\alpha \times \alpha \to \mathsf{Bool}) \times \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{f_{\mathsf{insert}}(n)}(\alpha)$. It is desirable to find closed forms for functions defined by such rewriting rules. In this work we are interested in the cases where closed-form solutions (or approximations of the solutions) are definable as indexed families of piecewise polynomials. For instance, a closed-form solution for f_{insert} is $\{n+i\}_{0 \leq i \leq 1}$.

The sets $TV(\tau)$ and $SV(\tau)$ of type and size variables of a type τ are defined inductively in the obvious way. Note that $SV(\mathsf{L}_0(\tau)) = \emptyset$, since all empty lists of the same underlying type represent the same data structure. For instance, $\mathsf{L}_0(\mathsf{L}_m(\mathsf{Int}))$ represents the same structure as $\mathsf{L}_0(\mathsf{L}_0(\mathsf{Int}))$.

Zero-order types without type variables and size variables are *ground types*:

$$\textit{GroundTypes} \quad \tau^\bullet ::= \tau \text{ such that } SV(\tau) = \emptyset \ \wedge \ TV(\tau) = \emptyset$$

The semantics of ground types is defined in Section 3.1. Here we give some examples: Int, $\mathsf{L}_5(\mathsf{Bool})$, $\mathsf{L}_{f_{\mathsf{insert}}(2)}(\mathsf{Bool})$ with $\mathcal{R} = \mathsf{Int}$ are ground types, whereas α, $\mathsf{L}_{n+5}(\mathsf{Int})$ and $\mathsf{L}_{f_{\mathsf{insert}}(n)}(\mathsf{Bool})$ with non-specified n are not. Examples of inhabitants of ground types are [True, True] and [False, True, True] for $\mathsf{L}_{f_{\mathsf{insert}}(2)}(\mathsf{Bool})$.

Let τ° denote a zero-order type where size expressions are all size variables or constants, like, e.g., $\mathsf{L}_n(\alpha)$. Function types are of the form:

$$\textit{FunctionTypes} \quad \tau^f ::= \tau_1^f \times \ldots \times \tau_{k'}^f \times \tau_1^\circ \times \ldots \times \tau_k^\circ \to \tau_0$$

where k' may be zero (i.e. the list $\tau_1^f, \ldots, \tau_{k'}^f$ is empty) and $SV(\tau_0)$ contains only size variables of $\tau_1^\circ, \ldots, \tau_k^\circ$. Consider, for instance, the function definition for $\mathsf{filter} : (\alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{f_{\mathsf{filter}}(n)}(\alpha)$

```
filter(g, l) =
  match l with | Nil ⇒ Nil
               | Cons(hd, tl) ⇒ if g(hd) then let l' = filter(g, tl) in Cons(hd, l')
                                else filter(g, tl)
```

The size function f_{filter} is defined by

$$\begin{array}{l} \vdash f_{\mathsf{filter}}(0) = 0 \\ n \geq 1 \vdash f_{\mathsf{filter}}(n) = 1 + f_{\mathsf{filter}}(n-1) \mid f_{\mathsf{filter}}(n-1) \end{array}$$

The closed-form solution for f_{filter} is $\{i\}_{0 \leq i \leq n}$.

A context Γ is a mapping from zero-order variables to zero-order types. A signature Σ is a mapping from function names to function types. The definition of SV is straightforwardly extended to contexts: $SV(\Gamma) = \bigcup_{\mathsf{z} \in dom(\Gamma)} SV(\Gamma(\mathsf{z}))$.

3.1 Semantics of Zero-order Types

In our semantic model, the purpose of the heap is to store lists. Therefore, a heap is a finite collection of locations ℓ that can store list elements. A location is the address of a cons-cell consisting of a head field *hd*, which stores a list element, and a tail field *tl*, which contains the location of the next cons-cell of the list, or the `NULL` address. Formally, a program value is either an integer or boolean constant, a location or the null address and a heap is a finite partial mapping from locations and fields into program values:

$$\begin{array}{lllll} \textit{Address} & \mathtt{adr} & ::= \ell \mid \mathtt{NULL} & \ell \in \textit{Loc} \\ \textit{Val} & v & ::= c \mid \mathtt{adr} & c \in \mathsf{Int} \cup \mathsf{Bool} \\ \textit{Heap} & h & : \quad \textit{Loc} \rightharpoonup \{hd, tl\} \rightharpoonup \textit{Val} & \end{array}$$

We will write $h.\ell.hd$ and $h.\ell.tl$ for the results of applications $h\ \ell\ hd$ and $h\ \ell\ tl$, which denote the values stored in the heap h at the location ℓ at its fields *hd* and *tl*, respectively. Let $h.\ell.[hd := v_h,\ tl := v_t]$ denote the heap equal to h everywhere but in ℓ, where at *hd* has the value v_h and at *tl* is v_t.

The semantics w of a program value v with respect to a specific heap h and a *ground type* $\tau^\bullet$ is a set-theoretic interpretation given via the four-place relation $v \models^h_{\tau^\bullet} w$. Integer and boolean constants interpret themselves, and locations are interpreted as non-cyclic lists:

$$
\begin{array}{lll}
c & \models^{h}_{\mathsf{Int} \cup \mathsf{Bool}} c & \\
\texttt{NULL} & \models^{h}_{\mathsf{L}_s(\tau^\bullet)} \quad \texttt{[]} & \textit{iff } 0 \in s \\
\ell & \models^{h}_{\mathsf{L}_s(\tau^\bullet)} \quad w_{hd} :: w_{tl} & \textit{iff } \ell \in dom(h) \wedge h.\ell.hd \models^{h|_{dom(h)\setminus\{\ell\}}}_{\tau^\bullet} w_{hd} \wedge \\
 & & \quad h.\ell.tl \models^{h|_{dom(h)\setminus\{\ell\}}}_{\mathsf{L}_{s-1}(\tau^\bullet)} w_{tl}
\end{array}
$$

where $h|_{dom(h)\setminus\{\ell\}}$ denotes the heap equal to h everywhere except in ℓ, where it is undefined.

It is easy to establish a natural connection between the size annotations in a ground list type and the length of a chain of cons-cells that "implements" its inhabitant in a heap. The length is defined by the function:

$$
\begin{array}{l}
\mathit{length} : \mathit{Heap} \rightharpoonup \mathit{Address} \rightharpoonup \mathcal{N} \\
\mathit{length}_h(\texttt{NULL}) = 0 \qquad \mathit{length}_h(\ell) = 1 + \mathit{length}_{h|_{dom(h)\setminus\{\ell\}}}(h.\ell.tl)
\end{array}
$$

Note that the function $\mathit{length}_h(-)$ does not take sharing into account, in the sense that the actual total size of allocated shared lists is less than the sum of their lengths. Thus, the sum of the lengths of the lists provides an upper bound on the amount of memory actually allocated.

Lemma 1 (Consistency of model relation).
The relation $\texttt{adr} \models^{h}_{\mathsf{L}_s(\tau^\bullet)} w$ *implies that* $\mathit{length}_h(\texttt{adr}) \in s$.

The proof is done by induction on the relation $\models$.

3.2 Operational Semantics of Program Expressions

The operational semantics is standard. It extends the semantics from [SvKvE07a] with higher-order functions.

We introduce a *frame store* as a mapping from program variables to program values. This mapping is maintained when a function body is evaluated. Before evaluation of the function body starts, the store contains only the actual parameters of the function. During evaluation, the store is extended with the variables introduced by pattern matching or let-constructs. These variables are eventually bound to the actual parameters. Thus there is no access beyond the current frame. Formally, a frame store s is a finite partial map from variables to values, $\mathit{Store}\ s\colon \mathit{ProgramVars} \rightharpoonup \mathit{Val}$.

Using heaps and a frame store and maintaining a mapping $\mathcal{C}$ of *closures*, from function names to the bodies of the function definitions, the operational semantics of program expressions is defined inductively in the usual way, as shown in Figure 1.

3.3 Typing Rules

A typing judgement is a relation of the form $D, \Gamma \vdash_\Sigma e : \tau$. Informally, it means that with the set of constraints D in the zero-order variable context Γ the expression e has type τ where the signature Σ contains type assumptions for all called functions. The set D of disequations and memberships is extended only

$$\frac{c \in \mathsf{Int} \cup \mathsf{Bool}}{s;\ h;\ \mathcal{C} \vdash c \rightsquigarrow c;\ h}\ \textsc{OSConst} \qquad \frac{}{s;\ h;\ \mathcal{C} \vdash \mathsf{z} \rightsquigarrow s(\mathsf{z});\ h}\ \textsc{OSVar}$$

$$\frac{}{s;\ h;\ \mathcal{C} \vdash \mathsf{Nil} \rightsquigarrow \mathtt{NULL};\ h}\ \textsc{OSNil}$$

$$\frac{s(\mathsf{hd}) = v_{hd} \qquad s(\mathsf{tl}) = v_{tl} \qquad \ell \notin dom(h)}{s;\ h;\ \mathcal{C} \vdash \mathsf{Cons(hd,\ tl)} \rightsquigarrow \ell;\ h[\ell.hd := v_{hd},\ \ell.tl := v_{tl}]}\ \textsc{OSCons}$$

$$\frac{s(\mathsf{x}) = \mathsf{True} \qquad s;\ h;\ \mathcal{C} \vdash e_1 \rightsquigarrow v;\ h'}{s;\ h;\ \mathcal{C} \vdash \mathsf{if\ x\ then}\ e_1\ \mathsf{else}\ e_2 \rightsquigarrow v;\ h'}\ \textsc{OSIfTrue}$$

$$\frac{s(\mathsf{x}) = \mathsf{False} \qquad s;\ h;\ \mathcal{C} \vdash e_2 \rightsquigarrow v;\ h'}{s;\ h;\ \mathcal{C} \vdash \mathsf{if\ x\ then}\ e_1\ \mathsf{else}\ e_2 \rightsquigarrow v;\ h'}\ \textsc{OSIfFalse}$$

$$\frac{s;\ h;\ \mathcal{C} \vdash e_1 \rightsquigarrow v_1;\ h_1 \qquad s[\mathsf{z} := v_1];\ h_1;\ \mathcal{C} \vdash e_2 \rightsquigarrow v;\ h'}{s;\ h;\ \mathcal{C} \vdash \mathsf{let\ z} = e_1\ \mathsf{in}\ e_2 \rightsquigarrow v;\ h'}\ \textsc{OSLet}$$

$$\frac{s(\mathsf{l}) = \mathtt{NULL} \qquad s;\ h;\ \mathcal{C} \vdash e_1 \rightsquigarrow v;\ h'}{\begin{array}{l}s;\ h;\ \mathcal{C} \vdash \mathsf{match\ l\ with}\ |\ \mathsf{Nil} \Rightarrow e_1 \\ \qquad\qquad\qquad\qquad\qquad\qquad |\ \mathsf{Cons(hd,\ tl)} \Rightarrow e_2\end{array} \rightsquigarrow v;\ h'}\ \textsc{OSMatch-Nil}$$

$$\frac{\begin{array}{c}h.s(\mathsf{l}).hd = v_{hd} \qquad h.s(\mathsf{l}).tl = v_{tl} \\ s[\mathsf{hd} := v_{hd}, \mathsf{tl} := v_{tl}];\ h;\ \mathcal{C} \vdash e_2 \rightsquigarrow v;\ h'\end{array}}{\begin{array}{l}s;\ h;\ \mathcal{C} \vdash \mathsf{match\ l\ with}\ |\ \mathsf{Nil} \Rightarrow e_1 \\ \qquad\qquad\qquad\qquad\qquad\qquad |\ \mathsf{Cons(hd,\ tl)} \Rightarrow e_2\end{array} \rightsquigarrow v;\ h'}\ \textsc{OSMatch-Cons}$$

$$\frac{s;\ h;\ \mathcal{C}[f := ((\mathsf{g}_1, \ldots, \mathsf{g}_{k'},\ \mathsf{z}_1, \ldots, \mathsf{z}_k) \times e_1)] \vdash e_2 \rightsquigarrow v;\ h'}{s;\ h;\ \mathcal{C} \vdash \mathsf{letfun\ f}(\mathsf{g}_1, \ldots, \mathsf{g}_{k'},\ \mathsf{z}_1, \ldots, \mathsf{z}_k) = e_1\ \mathsf{in}\ e_2 \rightsquigarrow v;\ h'}\ \textsc{OSLetFun}$$

$$\frac{\begin{array}{c}s(\mathsf{z}'_1) = v_1\ \ \ldots\ \ s(\mathsf{z}'_k) = v_k \\ \mathcal{C}(\mathsf{f}) = (\mathsf{g}_1, \ldots, \mathsf{g}_{k'},\ \mathsf{z}_1, \ldots, \mathsf{z}_k) \times e_f \\ [\mathsf{z}_1 := v_1, \ldots, \mathsf{z}_k := v_k];\ h;\ \mathcal{C} \vdash e_f[\mathsf{g}_1 := \mathsf{f}_1, \ldots, \mathsf{g}_{k'} := \mathsf{f}_{k'}] \rightsquigarrow v;\ h'\end{array}}{s;\ h;\ \mathcal{C} \vdash \mathsf{f}(\mathsf{f}_1, \ldots, \mathsf{f}_{k'},\ \mathsf{z}'_1, \ldots, \mathsf{z}'_k) \rightsquigarrow v;\ h'}\ \textsc{OSFunApp}$$

Fig. 1. Operational-semantics rules

in the pattern-matching rule. When the nil-branch is entered on a list $\mathsf{L}_s(\alpha)$, then D is extended with $0 \in s$. When the cons-branch is entered, then D is extended with $n' \geq 1,\ n' \in s$, where n' is a fresh size variable in D.

Given types $\tau = \mathsf{L}_{s_1}(\ldots \mathsf{L}_{s_k}(\alpha)\ldots)$ and $\tau' = \mathsf{L}_{s'_1}(\ldots \mathsf{L}_{s'_k}(\alpha)\ldots)$, let the entailment $D \vdash \tau \to \tau'$ abbreviate the collection of rules that (conditionally) rewrite s_i to s'_i:

$D \vdash s_1 \to s'_1$
$D \vdash s_2 \to s'_2$, if there exists a positive value in s'_1
$D \vdash s_3 \to s'_3$, if there exist positive values in s'_1 and s'_2
...
$D \vdash s_k \to s'_k$, if there exist positive values in each $s'_1, \ldots, s'_{k-1}$

For instance, the entailment $n \geq 2 \vdash \mathsf{L}_{f_1(n)}(\mathsf{L}_{f_2(n)}(\alpha)) \to \mathsf{L}_{n-1}(\mathsf{L}_{n^2}(\alpha))$ abbreviates the rules $n \geq 2 \vdash f_1(n) \to n-1$ and $\mathrm{n} \geq 2 \vdash f_2(n) \to n^2$. However, the entailment $n = 1 \vdash \mathsf{L}_{f_1(n)}(\mathsf{L}_{f_2(n)}(\alpha)) \to \mathsf{L}_{n-1}(\mathsf{L}_{n^2}(\alpha))$ abbreviates the single rule $n = 1 \vdash f_1(n) = n-1$. The rule $n = 1 \vdash f_2(n) \to n^2$ is not present because $f_1(1) = 0$ and thus the outer list must be empty.

If both rewriting rules $D \vdash s \to p_1$ and $D \vdash s \to p_2$ hold, then we abbreviate them by $D \vdash s \to p_1 \mid p_2$. This notation is lifted to types: $D \vdash \tau \to \tau_1 \mid \tau_2$ abbreviates $D \vdash \tau \to \tau_1$ and $D \vdash \tau \to \tau_2$. A set-theoretic semantics for conditional non-deterministic rewriting rules is given in technical report [SvET08a].

The typing judgement relation is defined by the following rules:

$$\frac{}{D,\ \Gamma \vdash_\Sigma c_i\colon \mathsf{Int}}\ \textsc{IConst} \qquad \frac{}{D,\ \Gamma \vdash_\Sigma c_b\colon \mathsf{Bool}}\ \textsc{BConst}$$

$$\frac{D \vdash \tau' \to \tau}{D,\ \Gamma,\ \mathsf{z}\colon \tau \vdash_\Sigma \mathsf{z}\colon \tau'}\ \textsc{Var} \qquad \frac{D \vdash \tau' \to \mathsf{L}_0(\tau)}{D,\ \Gamma \vdash_\Sigma \mathsf{Nil}\colon \tau'}\ \textsc{Nil}$$

$$\frac{D \vdash \tau' \to \mathsf{L}_{s+1}(\tau_2) \qquad D \vdash \tau_2 \to \tau_1}{D,\ \Gamma,\ \mathsf{hd}\colon \tau_1,\ \mathsf{tl}\colon \mathsf{L}_s(\tau_2) \vdash_\Sigma \mathsf{Cons}(\mathsf{hd}, \mathsf{tl})\colon \tau'}\ \textsc{Cons}$$

$$\frac{\begin{array}{c} D \vdash \tau \to \tau_1 \mid \tau_2 \\ \Gamma(\mathsf{x}) = \mathsf{Bool} \qquad D,\ \Gamma \vdash_\Sigma e_t\colon \tau_1 \qquad D,\ \Gamma \vdash_\Sigma e_f\colon \tau_2 \end{array}}{D,\ \Gamma \vdash_\Sigma \mathsf{if\ x\ then}\ e_t\ \mathsf{else}\ e_f\colon \tau}\ \textsc{If}$$

$$\frac{\mathsf{z} \notin dom(\Gamma) \qquad D,\ \Gamma \vdash_\Sigma e_1\colon \tau_z \qquad D,\ \Gamma,\ \mathsf{z}\colon \tau_z \vdash_\Sigma e_2\colon \tau}{D,\ \Gamma \vdash_\Sigma \mathsf{let\ z} = e_1\ \mathsf{in}\ e_2\colon \tau}\ \textsc{Let}$$

$$\frac{\begin{array}{c} D,\ 0 \in s,\ \Gamma,\ \mathsf{l}\colon \mathsf{L}_s(\tau) \vdash_\Sigma e_{\mathsf{Nil}}\colon \tau' \qquad \mathsf{hd}, \mathsf{tl} \notin dom(\Gamma) \\ D,\ n' \geq 1 \in s,\ \Gamma, \mathsf{hd}\colon \tau,\ \mathsf{l}\colon \mathsf{L}_s(\tau),\ \mathsf{tl}\colon \mathsf{L}_{s-1}(\tau) \vdash_\Sigma e_{\mathsf{Cons}}\colon \tau' \end{array}}{D;\ \mathsf{l}\colon \mathsf{L}_s(\tau) \vdash_\Sigma \begin{array}{l} \mathsf{match\ l\ with} \mid \mathsf{Nil} \Rightarrow e_{\mathsf{Nil}} \\ \qquad\qquad \mid \mathsf{Cons}(\mathsf{hd}, \mathsf{tl}) \Rightarrow e_{\mathsf{Cons}} \end{array} : \tau'}\ \textsc{Match}$$

where $n' \notin SV(D)$. Note that if in the MATCH-rule s is single-valued, then the statements in the nil and cons branches are $s = 0$ and $s \geq 1$, respectively.

$$\frac{\begin{array}{c}\Sigma(f) = \tau_1^f \times \ldots \times \tau_{k'}^f \times \tau_1^\circ \times \cdots \times \tau_k^\circ \to \tau_0 \\ \Sigma(\mathsf{g}_1) = \tau_1^f, \ldots, \Sigma(\mathsf{g}_{k'}) = \tau_{k'}^f \\ \mathsf{z}_1 : \tau_1^\circ, \ldots, \mathsf{z}_k : \tau_k^\circ \vdash_\Sigma e_1 : \tau_0 \qquad D;\ \Gamma \vdash_\Sigma e_2 : \tau'\end{array}}{D;\ \Gamma \vdash_\Sigma \mathsf{letfun}\ f(\mathsf{g}_1, \ldots, \mathsf{g}_{k'},\ \mathsf{z}_1, \ldots, \mathsf{z}_k) = e_1\ \mathsf{in}\ e_2 : \tau'}\ \textsc{LetFun}$$

$$\frac{\begin{array}{c}\Sigma(f) = \tau_1^f \times \ldots \times \tau_{k'}^f \times \tau_1^\circ \times \ldots \times \tau_k^\circ \to \tau_0 \\ \Sigma(\mathsf{g}_i) \text{ is an instance of the type schema} \tau_i^f \\ D \vdash \tau \to \sigma(\tau_0) \qquad D \vdash C(\tau_1, \ldots, \tau_k)\end{array}}{D, \Gamma, \mathsf{z}_1 : \tau_1, \ldots, \mathsf{z}_1 : \tau_k \vdash_\Sigma \mathsf{f}(\mathsf{g}_1, \ldots, \mathsf{g}_{k'},\ \mathsf{z}_1, \ldots, \mathsf{z}_k) : \tau}\ \textsc{FunApp}$$

The function application rule computes a substitution σ from the formal size and type variables to the actual size expressions and types, and a set C of equations collecting restrictions on the actual input types. These restrictions are of the form $\tau \equiv \tau'$ abbreviating equality of the corresponding underlying types and size functions. The equation $\tau \equiv \tau'$ belongs to C if τ and τ' are actual types corresponding to the same formal type. As an example of such an equivalence consider a call to a function $\mathsf{scalarprod} : \mathsf{L}_m(\mathsf{Int}) \times \mathsf{L}_m(\mathsf{Int}) \to \mathsf{Int}$. Due to the occurrence of m in both arguments the actual parameters $\mathsf{l} : \tau$ and $\mathsf{l}' : \tau'$ corresponding to the same formal type $\mathsf{L}_m(\mathsf{Int})$ must have equal sizes. To see how the substitution σ is applied to the whole type, consider a formal size parameter m with $\sigma(m) = s'$. Then,

$$\sigma(\mathsf{L}(\ldots \mathsf{L}_{s(m)}(\ldots \mathsf{L}(\alpha)\ldots)\ldots)) = \mathsf{L}(\ldots \mathsf{L}_{s(s')}(\ldots \mathsf{L}(\alpha)\ldots)\ldots)\ .$$

We write $s(n)$ when we want to emphasize that n if free in s.

Now we illustrate with an example how the typing rules are used to construct rewriting rules for multivalued size functions. Consider a function rel that produces all pairs of elements from two argument lists that are related to each other according to a given predicate. For instance $\mathsf{rel}(>,\ [2,\ 3,\ 5],\ [2,\ 4]) = [[3,\ 2],\ [5,\ 2],\ [5,\ 4]]$. This function calls an auxiliary function $\mathsf{rel\_pairs}$, that given a single element z and a list, produces the list of all pairs $(z,\ z')$ of the related elements, where z' runs over the list. The definitions for rel and $\mathsf{rel\_pairs}$ are:

$$\begin{array}{l}\mathsf{rel}(\mathsf{g}, \mathsf{l}_1, \mathsf{l}_2) = \mathsf{match}\ \mathsf{l}_1\ \mathsf{with}\ |\ \mathsf{Nil} \Rightarrow \mathsf{Nil} \\ \qquad\qquad\qquad\qquad\qquad\qquad |\ \mathsf{Cons}(\mathsf{hd},\ \mathsf{tl}) \Rightarrow \mathsf{append}(\mathsf{rel\_pairs}(\mathsf{g}, \mathsf{hd}, \mathsf{l}_2),\ \mathsf{rel}(\mathsf{g}, \mathsf{tl}, \mathsf{l}_2))\end{array}$$

$$\begin{array}{l}\mathsf{rel\_pairs}(\mathsf{g}, \mathsf{z}, \mathsf{l}) = \mathsf{match}\ \mathsf{l}\ \mathsf{with}\ |\ \mathsf{Nil} \Rightarrow \mathsf{Nil} \\ \qquad\qquad\qquad\qquad\qquad\qquad |\ \mathsf{Cons}(\mathsf{hd},\ \mathsf{tl}) \Rightarrow\ \mathsf{if}\ \mathsf{g}(\mathsf{z}, \mathsf{hd})\ \mathsf{then} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{Cons}(\mathsf{Cons}(\mathsf{z},\ \mathsf{Cons}(\mathsf{hd},\ \mathsf{Nil})), \mathsf{rel\_pairs}(\mathsf{g}, \mathsf{z}, \mathsf{tl})) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{rel\_pairs}(\mathsf{g},\ \mathsf{z},\ \mathsf{tl})\end{array}$$

The types are $(\alpha \to \alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{f_{\mathsf{rel}_1}}(\mathsf{L}_{f_{\mathsf{rel}_2}}(\alpha))$ and $(\alpha \to \alpha \to \mathsf{Bool}) \times \alpha \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{f_{\mathsf{rel\_pairs}_1}}(\mathsf{L}_{f_{\mathsf{rel\_pairs}_2}}(\alpha))$, respectively. We want to construct rewriting rules for $f_{\mathsf{rel\_pairs}_1}$ and $f_{\mathsf{rel\_pairs}_2}$. We apply typing rules in the backward style to the body of $\mathsf{rel\_pairs}$. For the sake of convenience, below in the typing judgements, we list only the relevant variables of the context.

1. We want to infer $f_{\mathsf{rel\_pairs}_1}$ and $f_{\mathsf{rel\_pairs}_2}$ such that

   $\mathsf{z}\colon \alpha, \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma e_{\mathsf{rel\_pairs}}\colon \mathsf{L}_{f_{\mathsf{rel\_pairs}_1}}(\mathsf{L}_{f_{\mathsf{rel\_pairs}_2}}(\alpha))$

2. We start applying the match-rule since $e_{\mathsf{rel\_pairs}}$ is a pattern-matching. We obtain

   $\mathsf{Nil}\text{-branch:}\quad n = 0 \vdash_\Sigma \mathsf{Nil}\colon \mathsf{L}_{f_{\mathsf{rel\_pairs}_1}}(\mathsf{L}_{f_{\mathsf{rel\_pairs}_2}}(\alpha))$
   $\mathsf{Cons}\text{-branch:}\ n \geq 1;\ \mathsf{z}\colon \alpha, \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma e'\colon \mathsf{L}_{f_{\mathsf{rel\_pairs}_1}}(\mathsf{L}_{f_{\mathsf{rel\_pairs}_2}}(\alpha))$

   where e' is the if-expression in the cons-branch.

3. Since the expression in the nil-branch is just Nil, we apply the nil-rule and obtain $n = 0 \vdash \mathsf{L}_{f_{\mathsf{rel\_pairs}_1}}(\mathsf{L}_{f_{\mathsf{rel\_pairs}_2}}(\alpha)) \to \mathsf{L}_0(\tau_0)$ that according to the definition of $D \vdash \tau \to \tau'$ reduces to $n = 0 \vdash f_{\mathsf{rel\_pairs}_1}(n) \to 0$.

4. Apply the if-rule to the expression e' in the cons-branch to obtain that $\mathsf{L}_{f_{\mathsf{rel\_pairs}_1}}(\mathsf{L}_{f_{\mathsf{rel\_pairs}_2}}(\alpha)) \to \tau_1 \mid \tau_2$, where

   $n \geq 1;\ \mathsf{z}\colon \alpha, \mathsf{hd}\colon \alpha, \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{Cons}(\mathsf{Cons}(\mathsf{z}, \mathsf{Cons}(\mathsf{hd}, \mathsf{Nil})), \mathsf{rel\_pairs}(\mathsf{g}, \mathsf{z}, \mathsf{tl}))\colon \tau_1$
   $n \geq 1;\ \mathsf{z}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{rel\_pairs}(\mathsf{g}, \mathsf{z}, \mathsf{tl})\colon \tau_2$

   Note, that the expression in the true-branch abbreviates the chain of let-bindings:

   $\mathsf{let}\ \mathsf{z}_1 = \mathsf{Nil}\ \mathsf{in}\ \mathsf{let}\ \mathsf{z}_2 = \mathsf{Cons}(\mathsf{hd}, \mathsf{z}_1)\ \mathsf{in}\ \mathsf{let}\ \mathsf{z}_3 = \mathsf{Cons}(\mathsf{z}, \mathsf{z}_2)\ \mathsf{in}$
   $\quad \mathsf{let}\ \mathsf{z}_4 = \mathsf{rel\_pairs}(\mathsf{g}, \mathsf{z}, \mathsf{tl})\ \mathsf{in}\ \mathsf{Cons}(\mathsf{z}_3, \mathsf{z}_4)$

   Let $e_{\mathsf{body}_1}, \ldots, e_{\mathsf{body}_4}$ denote the let-bodies corresponding to the let-bindings of $\mathsf{z}_1, \ldots, \mathsf{z}_4$, respectively.

5. Applying the let-rule to z_1-binding gives

   $\mathsf{let}_1:\quad n \geq 1 \vdash_\Sigma \mathsf{Nil}\colon ?\tau^1$
   $\mathsf{body}_1: n \geq 1;\ \mathsf{z}_1\colon ?\tau^1, \ldots \vdash_\Sigma e_{\mathsf{body}_1}\colon \tau_1$

6. Applying the nil-rule to the let-branch instantiates $?\tau^1$ with $\mathsf{L}_0(?\tau^{10})$, so we obtain $n \geq 1;\ \mathsf{z}_1\colon \mathsf{L}_0(?\tau^{10}), \ldots \vdash_\Sigma e_{\mathsf{body}_1}\colon \tau_1$.

7. Applying the let-rule to z_2-binding gives

   $\mathsf{let}_2:\quad n \geq 1; \mathsf{hd}\colon \alpha, \mathsf{z}_1\colon \mathsf{L}_0(?\tau^{10}) \vdash_\Sigma \mathsf{Cons}(\mathsf{hd}, \mathsf{z}_1)\colon ?\tau^2$
   $\mathsf{body}_2: n \geq 1;\ \mathsf{z}_2\colon ?\tau^2, \ldots \vdash_\Sigma e_{\mathsf{body}_2}\colon \tau_1$

8. Applying the cons-rule to the let-branch instantiates $?\tau^{10}$ with α and $?\tau^2$ with $\mathsf{L}_1(\alpha)$, so we obtain $\mathsf{body}_2:\ n \geq 1;\ \mathsf{z}\colon \alpha,\ \mathsf{z}_2\colon \mathsf{L}_1(\alpha), \ldots \vdash_\Sigma e_{\mathsf{body}_2}\colon \tau_1$.

9. Similarly, applying the let- and cons-rules for z_3-binding gives

   $\mathsf{body}_3:\ n \geq 1;\ \mathsf{z}\colon \alpha, \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha),\ \ \mathsf{z}_3\colon \mathsf{L}_2(\alpha) \vdash_\Sigma e_{\mathsf{body}_3}\colon \tau_1$

10. Applying the let- and funapp-rules for z_4-binding gives

    $\mathsf{body}_4:\ n \geq 1;\ \mathsf{z}_3\colon \mathsf{L}_2(\alpha), \mathsf{z}_4\colon \mathsf{L}_{f_{\mathsf{rel\_pairs}_1}(n-1)}(\mathsf{L}_{f_{\mathsf{rel\_pairs}_2}(n-1)}(\alpha)) \vdash_\Sigma \mathsf{Cons}(\mathsf{z}_3, \mathsf{z}_4)\colon \tau_1$

11. Applying the cons-rule gives $n \geq 1 \vdash \tau_1 \to \mathsf{L}_{f_{\mathsf{rel\_pairs}_1}(n-1)+1}(\mathsf{L}_{f_{\mathsf{rel\_pairs}_2}(n-1)}(\alpha))$ and $n \geq 1 \vdash f_{\mathsf{rel\_pairs}_2}(n-1) \to 2$.

12. Applying the function application rule in the false-branch gives $n \geq 1 \vdash \tau_2 \to \mathsf{L}_{f_{\mathsf{rel\_pairs}_1}(n-1)}(\mathsf{L}_{f_{\mathsf{rel\_pairs}_2}(n-1)}(\alpha))$.

13. Recalling the multiple-choice-rewriting side condition from the application of the if-rule we obtain

$$n \geq 1 \vdash \mathsf{L}_{f_{\mathsf{rel\_pairs}_1}(n)}(\mathsf{L}_{f_{\mathsf{rel\_pairs}_2}(n)}(\alpha)) \to \mathsf{L}_{f_{\mathsf{rel\_pairs}_1}(n-1)+1}(\mathsf{L}_{f_{\mathsf{rel\_pairs}_2}(n-1)}(\alpha)) \mid \mathsf{L}_{f_{\mathsf{rel\_pairs}_1}(n-1)}(\mathsf{L}_{f_{\mathsf{rel\_pairs}_2}(n-1)}(\alpha))$$

thus we have the following rewriting rules corresponding to the nil-branch and the true and false cases of the cons-branch, respectively.

$$\begin{array}{ll} n \geq 1 & \vdash f_{\mathsf{rel\_pairs}_1}(n) \to f_{\mathsf{rel\_pairs}_1}(n-1)+1 \mid f_{\mathsf{rel\_pairs}_1}(n-1) \\ n' \in f_{\mathsf{rel\_pairs}_1}(n-1)+1,\ n' \geq 1,\ n \geq 1 & \vdash f_{\mathsf{rel\_pairs}_2}(n) \to f_{\mathsf{rel\_pairs}_2}(n-1) \\ n' \in f_{\mathsf{rel\_pairs}_1}(n-1),\ n' \geq 1,\ n \geq 1 & \vdash f_{\mathsf{rel\_pairs}_2}(n) \to f_{\mathsf{rel\_pairs}_2}(n-1) \end{array}$$

Recall that $\vdash f_{\mathsf{rel\_pairs}_1}(0) \to 0$ and, from, $n \geq 1 \vdash f_{\mathsf{rel\_pairs}_2}(n-1) \to 2$ due to the nil-rule in the nil-branch and the last cons-rule (step 11), respectively. Therefore, we obtain

$$\begin{array}{l} \vdash f_{\mathsf{rel\_pairs}_1}(0) \to 0 \\ n \geq 1 \vdash f_{\mathsf{rel\_pairs}_1}(n) \to f_{\mathsf{rel\_pairs}_1}(n-1)+1 \mid f_{\mathsf{rel\_pairs}_1}(n-1) \\ n \geq 0 \vdash f_{\mathsf{rel\_pairs}_2}(n) \to 2 \end{array}$$

Similarly we obtain rewriting rules for the multivalued size functions for rel.

$$\begin{array}{l} \vdash f_{\mathsf{rel}_1}(0,m) \to 0 \\ n \geq 1 \vdash f_{\mathsf{rel}_1}(n,m) \to f_{\mathsf{rel\_pairs}_1}(m) + f_{\mathsf{rel}_1}(n-1,m) \\ n \geq 1 \vdash f_{\mathsf{rel}_2}(n,m) = f_{\mathsf{rel\_pairs}_2}(m) \end{array}$$

The equality in the last entailment comes from the type equality collected in the C set of the FunApp rule.

3.4 Semantics of Typing Judgements (Soundness)

The set-theoretic semantics of typing judgements is formalised later in this section as the soundness theorem, which is defined by means of the following two predicates. One indicates if a program value is *valid* with respect to a certain heap and a ground type. The other does the same for sets of values and types, taken from a frame store and a ground context $\Gamma^\bullet$:

$$\begin{array}{ll} \mathit{Valid}_{\mathsf{val}}(v, \tau^\bullet, h) & = \exists_w [\ v \models^h_{\tau^\bullet} w\ ] \\ \mathit{Valid}_{\mathsf{store}}(\mathit{vars}, \Gamma^\bullet, s, h) & = \forall_{\mathsf{z} \in \mathit{vars}} [\ \mathit{Valid}_{\mathsf{val}}(s(\mathsf{z}), \Gamma^\bullet(\mathsf{z}), h)\ ] \end{array}$$

Let a valuation ϵ map size variables to concrete sizes (numbers from the ring $\mathcal{R}$) and an instantiation η map type variables to ground types:

$$\begin{array}{ll} \mathit{Valuation} & \epsilon : \mathit{SizeVariables} \to \mathcal{R} \\ \mathit{Instantiation} & \eta : \mathit{TypeVariables} \to \tau^\bullet \end{array}$$

Now, stating the soundness theorem is straightforward. Informally, it states that assuming that the context zero-order variables are *valid*, i.e. indeed point to lists of the sizes mentioned in the input types, then the result in the heap will be *valid*, i.e. of the size indicated in the output type. The proof is given in Section 2 of the appendix.

Theorem 1 (Soundness). *For any store s, heaps h and h', closure $\mathcal{C}$, expression e, value v, context Γ, quantifier-free formula D, signature Σ, type τ, size valuation ϵ, and type instantiation η such that*

- *the expression e terminates with the value v, i.e. in terms of operational semantics the relation $s; h; \mathcal{C} \vdash e \rightsquigarrow v; h'$ holds,*
- *$D, \Gamma \vdash_\Sigma e : \tau$ is a node in the derivation tree for some function body,*
- *$dom(s) = dom(\Gamma)$,*
- *$D(\epsilon(\overline{n}))$ holds, where $\overline{n}$ is the set of size variables from $dom(\Gamma \cup D)$,*
- *$Valid_{\mathsf{store}}(dom(s), \eta(\epsilon(\Gamma)), s, h)$ holds,*

then the return value v is valid according to its return type τ, i.e.

$$Valid_{\mathsf{val}}(v, \eta(\epsilon(\tau)), h')$$

holds.

4 Approximation of Multivalued Size Functions

In practice, size functions in closed forms, like $f(n) = \{n, n+1\}$ for insert, are preferable to ones in the form of rewriting rules. However, inference of closed forms is a hard problem. Instead, we propose to infer their set-theoretic approximations given by indexed families of piecewise polynomials.

Definition. *A family $\{g(\overline{n}, \overline{i})\}_{Q(\overline{n},\overline{i})}$ of piecewise polynomials, where $Q(\overline{n}, \overline{i})$ is a quantifier-free first-order arithmetic predicate, approximates a multivalued function s if and only if for all $\overline{n}$ in the domain of s, $s(\overline{n}) \subseteq \{g(\overline{n}, \overline{i})\}_{Q(\overline{n},\overline{i})}$.* In other words, for all $m \in s(\overline{n})$, there exists $\overline{i}$ such that $m = g(\overline{n}, \overline{i})$ and the predicate $Q(\overline{n}, \overline{i})$ holds.

Given a multivalued size function in the form of rewriting rules, the inference procedure first generates a candidate approximating family and then checks whether it indeed approximates the function.

4.1 Inferring a Candidate Approximating Family of Polynomials

To give an idea behind the interactive procedure that infers approximating families of piecewise polynomials, we start with a simple example. We show how to infer candidate polynomial lower and upper bounds for the size function of insert and how to construct an approximating family from it. Recall the size rewriting system for insert:

$$\begin{aligned} &\vdash f_{\mathsf{insert}}(0) \to 1 \\ n \geq 1 &\vdash f_{\mathsf{insert}}(n) \to n \mid 1 + f_{\mathsf{insert}}(n-1) \end{aligned}$$

Assume that $p_{\min}$ and $p_{\max}$ are linear, that is, that they are of the form $a_{\min}n + b_{\min}$ and $a_{\max}n + b_{\max}$, respectively. We want to find the coefficients $a_{\min}$, $b_{\min}$, $a_{\max}$, $b_{\max}$ (as we did in [SvKvE07a] for strict polynomial (single-valued) size functions, where the lower and upper bounds were equal). To reconstruct $p_{\min}$, one needs to know two points on its graph, and the same holds

for $p_{\max}$. Take $n = 1$ and $n = 2$. Evaluating the rewriting rules gives $f_{\mathsf{insert}}(1) = \{1,\ 2\}$ and $f_{\mathsf{insert}}(2) = \{2,\ 3\}$. Pick up the minimal values from $f_{\mathsf{insert}}(1)$ and $f_{\mathsf{insert}}(2)$ and assume that the graph of $p_{\min}$ contains the points (1, 1) and (2, 2). Similarly, pick up the maximal values from $f_{\mathsf{insert}}(1)$ and $f_{\mathsf{insert}}(2)$ and assume that $p_{\max}$ contains (1, 2) and (2, 3). We obtain two systems of equations, for $a_{\min}$, $b_{\min}$ and $a_{\max}$, $b_{\max}$, respectively:

$$\begin{cases} a_{\min} + b_{\min} = 1 \\ 2a_{\min} + b_{\min} = 2 \end{cases} \qquad \begin{cases} a_{\max} + b_{\max} = 2 \\ 2a_{\max} + b_{\max} = 3 \end{cases}$$

Solving these linear systems we get $a_{\min} = 1$, $b_{\min} = 0$ and $a_{\max} = 1$, $b_{\max} = 1$. Thus, we reconstruct the expressions for $p_{\min}(n) = n$ and $p_{\max}(n) = n + 1$, and the approximating family $p_{\min}(n) + i$, where $0 \le i \le \delta(n)$ with $\delta(n) = p_{\max}(n) - p_{\min}(n) = 1$. The rest of the job is to check whether this reconstruction approximates the solution of the rewriting rules. We discuss this in Section 4.2.

It is easy to see that we have inferred accurate bounds for insert, i.e. the *greatest lower* and the *lowest upper* bounds for the multivalued size function. Moreover, given any $n \ge 1$, there is an evaluation path for $f_{\mathsf{insert}}(n)$ that evaluates to $p_{\min}(n)$, and there is a path that evaluates to $p_{\max}(n)$. It explains the choice of the *step*=1: it is enough to take two consecutive natural numbers to generate the systems of equations for the coefficients of the linear lower and upper bounds.

The bounds for insert are one-variable and the systems of linear equations w.r.t. the polynomial coefficients are trivially consistent if one chooses different testing size values, in the example $n = 1$ and $n = 2$. The reason for this is that the matrix of such a system has a 1-variable non-zero *Vandermonde* determinant. In the multivariate case, say s variables, the consistency of the systems for $p_{\min}$ and $p_{\max}$ (for which the corresponding multivariate Vandermonde determinant is non-zero) depends on a more involving condition. If the testing values, i.e. the points in an s-dimensional space, lie in a so called *Node Configuration A* (**NCA** configuration [CL87]), the systems for $p_{\min}$ and $p_{\max}$ have unique solutions, and thus the polynomials are uniquely defined.

We describe an **NCA** configuration for the case $s = 2$ in detail. Let d be the degree of a polynomial and $N_d^2 = \binom{d+2}{2}$ denote the amount of its coefficients. A set W of N_d^2 points on a plane lie in a *2-dimensional* ***NCA*** *configuration* if there exist lines $\gamma_1, \ldots, \gamma_{d+1}$ in the space $\mathcal{R}^2$, such that $d + 1$ points of W lie on γ_{d+1}, d points of W lie on $\gamma_d \setminus \gamma_{d+1}$, ..., and finally 1 point of W lies on $\gamma_1 \setminus (\gamma_2 \cup \ldots \cup \gamma_{d+1})$. The simplest example of an **NCA** configuration on a plane is a "triangle" of points, where $d+1$ different points lie on the line $y = 1$, d points lie on the line $y = 2$,..., and 1 point lies on the line $y = d + 1$. For instance, with $d = 2$ a two variable polynomial has $N_2^2 = \binom{2+2}{2} = 6$ coefficients, hence we pick up 6 points: $(1, 1)$, $(2, 1)$, $(3, 1)$, $(1, 2)$, $(2, 2)$ and $(1, 3)$.

For dimensions $s > 2$ this configuration is formulated inductively, using the notion of a hyperplane [CL87]. Since the definition itself is technically involved, we just give an example of an **NCA** for 3 variables ($s = 3$) and degree $d = 2$. To define a polynomial of three variables of degree 2 one needs to know $N_2^3 = \binom{2+3}{3} = 10$ coefficients, hence we need to place 10 points:

1. on the plane $x = 0$ take the "triangle" of $N_2^2 = 6$ points that lies in the 2-dimensional **NCA**, say (0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 1, 0), (0, 1, 1), (0, 2, 0),
2. on the plane $x = 1$ take the "triangle" of $N_1^2 = 3$ points that lies in the 2-dimensional **NCA**, say (1, 0, 0), (1, 0, 1), (1, 1, 0),
3. on the plane $x = 2$ take the point (2, 0, 0).

Now we give a general procedure for inferring lower and upper polynomial bounds from a given system of size rewriting rules.

INPUT: The degrees $d_{\min}$, $d_{\max}$ of hypothetical upper and lower bounds, s size variables, $\overline{n} = (n_1, .., n_s)$, initial test points $w^0_{\min} = \overline{n}^0_{\min}$, $w^0_{\max} = \overline{n}^0_{\max}$, steps $\epsilon_{\min}$, $\epsilon_{\max}$ and the system G of size rewriting rules.

OUTPUT: A lower $p_{\min}$ and an upper $p_{\max}$ bound or the proposal to repeat the procedure for higher degrees and/or other $w^0_{\min}$, $w^0_{\max}$, $\epsilon_{\min}$, $\epsilon_{\max}$.

PROCEDURE:

1. According to the initial points and steps, pick up N_d^s points $w = (n_1, \ldots, n_s)$ in the s-dimensional space that lie in **NCA** configuration; let they constitute the sets $W_{\min}$. Similarly, generate $W_{\max}$.
2. For any $w^i \in W_{\min}$ compute the set $f_{i,\min} = f(w^i)$. Similarly, compute $f_{j,\max}$ for any $w^j \in W_{\max}$.
3. For any f_i (f_j) pick up its minimal $f_i^{\min}$ (maximal $f_j^{\max}$) value.
4. Interpolate $p_{\min}$ using the points $(w_i, f_i^{\min})$ by solving the system of linear equations w.r.t. its coefficients. Also interpolate $p_{\max}$ using the points $(w_j, f_j^{\max})$ by solving the system of linear equations w.r.t. its coefficients.
5. Check whether the family $\{p_{\min}(\overline{n}) + i\}_{0 \le i \le (p_{\max}(\overline{n}) - p_{\min}(\overline{n}))}$ approximates the multivalued function defined by G. If it does, stop and output $p_{\min}$ and $p_{\max}$. Otherwise pick up other parameters d, $w^0_{\min}$, $w^0_{\max}$, $\epsilon_{\min}$, $\epsilon_{\max}$

The choice of the parameters $w^0_{\min}$, $w^0_{\max}$, $\epsilon_{\min}$, $\epsilon_{\max}$ is crucial. Based on them, the procedure generates the points $(w, f(w))$. A bad choice of parameters has one of two consequences: either no bounds will be detected even if they exist, or loose bounds will be inferred. The first happens when $W_{\min}$ (resp., $W_{\max}$) are constructed in such a way that there is no bound $p_{\min}$ (resp., $p_{\max}$) such that its graph contains all points from $(w_{\min}, f_i^{min})$ (resp., $(w_{\max}, f_j^{max})$). Consider, for instance, a function $\mathsf{divtwo}\colon \mathsf{L}_n(\alpha) \to \mathsf{L}_{f(n)}(\alpha)$ that takes a list of length n and returns a list of length $n/2$ if n is even, and $(n-1)/2$, if n is odd. The rewriting rules for the size function are $f(0) \to 0$, $f(1) \to 0$, $n \ge 2 \vdash f(n) \to f(n-2) + 1$. Take $d = 1$, $n^0_{\min,\max} = 0$, $\epsilon_{\min,\max} = 1$. Then $f(0) = f(1) = 0$. There is no linear upper bound that contains both, $(0, 0)$ and $(1, 0)$, points since output type $\mathsf{L}_0(\alpha)$ is rejected by the checker. Still, linear bounds can be obtained if suitable parameters are provided. Take e.g. $n^0_{\min} = 3$, $n^0_{\max} = 2$, $\epsilon_{\min,\max} = 2$. Then $f(3) = 1$, $f(5) = 2$ and $p_{\min}(n) = (n-1)/2$, similarly $f(2) = 1$, $f(4) = 2$ and $p_{\max}(n) = n/2$.

Rough lower (upper) bound are obtained when the graph of some lower (upper) bound does contain all points $(w_i, f_i^{\min})$ with $w_i \in W_{\min}$ (resp., $(w_j, f_j^{\max})$ with $w_j \in W_{\max}$), but the bound itself is rough. For instance, this happens when $n^0 = 0$ for insert. Then $f(0) = 1$, $f(1) = \{1, 2\}$, so the inferred $p_{\min}(n) = 1$.

The examples above show that users should choose the parameters based on common sense and their intuitive knowledge about the functions under considerations. We recommend not to include the base-of-recursion sizes into sets of test points since these cases are usually "non-typical".

Adaptations for inferring families of piecewise polynomials are possible. The user hints the inference system on which areas P_i she assumes different pieces of polynomial bounds. Different parameters must be provided for each piece.

As a more elaborated example, consider the inference procedure for the function rel (defined in Section 3.3). The inferred size rewriting system is:

$$\begin{aligned} &\vdash f_{\mathsf{rel}_1}(0, m) \to 0 \\ n \geq 1 &\vdash f_{\mathsf{rel}_1}(n, m) \to f_{\mathsf{rel\_pairs}_1}(m) + f_{\mathsf{rel}_1}(n-1, m) \\ n \geq 1 &\vdash f_{\mathsf{rel}_2}(n, m) = f_{\mathsf{rel\_pairs}_2}(m) \end{aligned}$$

We show how to infer the family $\{i\}_{0 \leq i \leq nm}$. A quadratic polynomial $q(n, m) = a_{20}n^2 + a_{02}m^2 + a_{11}nm + a_{10}n + a_{01}m + a_{00}$ of two variables has 6 coefficients, so to define the polynomial one needs to know 6 points (n_i, m_i, q_i) on the graph of q. The coefficients are computed as the solution of the system of linear equations $q_i = a_{20}n_i^2 + a_{02}m_i^2 + a_{11}n_i m_i + a_{10}n_i + a_{01}m_i + a_{00}$, where $1 \leq i \leq 6$. For instance, one can take the points (n, m) from $\{(1,1), (2,1), (3,1), (1,2), (2,2), (1,3)\}$. Then, the linear system with respect to the coefficients of q has the form

$$\begin{aligned} a_{20} + a_{02} + a_{11} + a_{10} + a_{01} + a_{00} &= q(1,1) \\ 4a_{20} + a_{02} + 2a_{11} + 2a_{10} + a_{01} + a_{00} &= q(2,1) \\ 9a_{20} + 3a_{02} + 3a_{11} + 3a_{10} + a_{01} + a_{00} &= q(3,1) \\ a_{20} + 4a_{02} + 2a_{11} + a_{10} + 2a_{01} + a_{00} &= q(1,2) \\ 4a_{20} + 4a_{02} + 4a_{11} + 2a_{10} + 2a_{01} + a_{00} &= q(2,2) \\ a_{20} + 9a_{02} + 3a_{11} + a_{10} + 3a_{01} + a_{00} &= q(1,3) \end{aligned}$$

To reconstruct $p_{\min}$ and $p_{\max}$, consider all possible evaluation paths for f_{rel} at these points, using the fact that for any fixed n, m there is only finite number of indices j satisfying $0 \leq j \leq m$. Recall that $\mathsf{rel\_pairs}_1(m) = \{j\}_{0 \leq j \leq m}$.

$$\begin{aligned} f_{\mathsf{rel}_1}(1,1) &= j + 0 &&= \{0,\ 1\} \\ f_{\mathsf{rel}_1}(2,1) &= j + f_{\mathsf{rel}_1}(1,\ 1) &&= \{0,\ 1,\ 2\} \\ f_{\mathsf{rel}_1}(3,1) &= j + f_{\mathsf{rel}_1}(2,\ 1) &&= \{0,\ 1,\ 2,\ 3\} \\ f_{\mathsf{rel}_1}(1,2) &= j + 0 &&= \{0,\ 1,\ 2\} \\ f_{\mathsf{rel}_1}(2,2) &= j + f_{\mathsf{rel}_1}(1,\ 2) &&= \{0,\ 1,\ 2,\ 3,\ 4\} \\ f_{\mathsf{rel}_1}(1,3) &= j + 0 &&= \{0,\ 1,\ 2,\ 3\} \end{aligned}$$

Thus, for the coefficients of $p_{\max}$ one has the system

$$
\begin{aligned}
a_{20} + a_{02} + a_{11} + a_{10} + a_{01} + a_{00} &= 1\\
4a_{20} + a_{02} + 2a_{11} + 2a_{10} + a_{01} + a_{00} &= 2\\
9a_{20} + 3a_{02} + 3a_{11} + 3a_{10} + a_{01} + a_{00} &= 3\\
a_{20} + 4a_{02} + 2a_{11} + a_{10} + 2a_{01} + a_{00} &= 2\\
4a_{20} + 4a_{02} + 4a_{11} + 2a_{10} + 2a_{01} + a_{00} &= 4\\
a_{20} + 9a_{02} + 3a_{11} + a_{10} + 3a_{01} + a_{00} &= 3
\end{aligned}
$$

The solution is $(0,0,1,0,0,0)$, so $p_{\max}(n,m) = nm$. The system for $p_{\min}$ has all zeros on its right hand side, thus $p_{\min} = 0$. The inferred family is indeed $\{i\}_{0\le i\le nm}$, which approximates the multivalued size function f_{rel_1}.

4.2 Checking Whether a Family Approximates a Size Function

Checking an inferred family is similar to type checking types annotated with families of piecewise polynomials directly [SvET08b]. In that type system, for instance, the output type of insert is $\mathsf{L}_{n+i}^{0\le i\le 1}(\alpha)$.

We show that, given a multivalued size function and some indexed family of piecewise polynomials, there is a set of first-order arithmetic entailments such that their satisfiability implies that the family approximates the size function. Such predicates are obtained by substituting indexed families of polynomials, which are to be checked as approximations, for the corresponding multivalued-function symbols in the rewriting rules. For instance, verifying whether the family $\{n+i\}_{0\le i\le 1}$ approximates $f_{\mathsf{insert}}(n)$ reduces to checking the entailments

$$
\begin{aligned}
&n = 0 && \vdash 1 = n{+}?i \wedge 0 \le ?i \le 1\\
&n \ge 1 && \vdash n = n{+}?i \wedge 0 \le ?i \le 1\\
&n \ge 1,\ 0 \le j \le 1 && \vdash 1 + ((n-1) + j) = n{+}?i \wedge 0 \le ?i \le 1
\end{aligned}
$$

corresponding to the nil-branch, and the true and false cases of the cons-branch, respectively. Checking succeeds by instantiating $?i$ to 1, 0 and j, respectively.

Substitution of an indexed family of polynomials $\{g(\overline{n},\overline{i})\}_{Q(\overline{n},\overline{i})}$ for a multivaluedfunction symbol f is defined in the usual way. Let an arithmetic expression $\varepsilon(\overline{n},\overline{i})$ contain size variables $\overline{n}$, indices $\overline{i}$ (such that $Q(\overline{n},\overline{i})$ holds), symbols $+$, $-$, $*$ and symbols of multivalued functions. Examples of $\varepsilon(\overline{n},\overline{i})$ are $1 + f_{\mathsf{insert}}(n-1)$, $f_{\mathsf{insert}}(n-2) + f_{\mathsf{insert}}(n-1)$ and $f_{\mathsf{rinsert}}(m-1,n) + i$, with $0 \le i \le 1$, where rinsert inserts a whole list by recursively calling insert. Substituting the family $\{n+i\}_{0\le i\le 1}$, for f_{insert} in the first expression results in $1+(n-1)+i = n+i$. In the second expression it gives $(n-2)+i_1+(n-1)+i_2 = 2n-3+i_1+i_2$, where $0 \le i_1, i_2 \le 1$. The substitution $\{n+j\}_{0\le j\le m}$, for $f_{\mathsf{rinsert}}(m-1,n)$ in the third expression results in $n+j+i$ with $0 \le j \le m-1$ and $0 \le i \le 1$. The appendix of [SvET08a] describes these examples in more detail.

We generalise substitution, denote by $[-]$, to types of the form annotated by indexed families of expressions, i.e., $\tau = \mathsf{L}_{\varepsilon_1}^{Q_1(\overline{n},\overline{i}_1)}(\ldots \mathsf{L}_{\varepsilon_k}^{Q_k(\overline{n},\overline{i}_k)}(\alpha)\ldots)$ in the natural way: $[\tau] = \mathsf{L}_{[\varepsilon_1]}^{Q_1'(\overline{n},\overline{i}_1,\overline{j}_1)}(\ldots \mathsf{L}_{[\varepsilon_k]}^{Q_k'(\overline{n},\overline{i}_k,\overline{j}_k)}(\alpha)\ldots)$.

To construct predicates to check candidate approximations, one also needs the notion of subtyping for types annotated by indexed families of piecewise

polynomials directly [SvET08b]. Examples of subtypings in those type system are $\vdash \mathsf{L}_{n+i}^{0\le i\le 1}(\alpha) \preccurlyeq \mathsf{L}_{n+i}^{0\le i\le 2}(\alpha)$ and $n = 0 \vdash \mathsf{L}_n(\mathsf{L}_2(\alpha)) \preccurlyeq \mathsf{L}_n(\mathsf{L}_i^{0\le i\le 2}(\alpha))$. Let $T = \mathsf{L}_{g^1(\overline{n},\overline{i}^1)}^{Q^1(\overline{n},\overline{i}^1)}(\ldots \mathsf{L}_{g^k(\overline{n},\overline{i}^k)}^{Q^k(\overline{n},\overline{i}^k)}(\alpha)\ldots)$ and $T' = \mathsf{L}_{g'^1(\overline{n},\overline{j}^1)}^{Q'^1(\overline{n},\overline{j}^1)}(\ldots \mathsf{L}_{g'^k(\overline{n},\overline{j}^k)}^{Q'^k(\overline{n},\overline{j}^k)}(\alpha)\ldots)$. Then $D \vdash T' \preccurlyeq T$ holds if and only if

$$\forall\, \overline{n}\,\overline{j}^1.\ D(\overline{n}) \wedge Q'^1(\overline{n},\overline{j}^1) \Longrightarrow\ \exists\, \overline{i}^1.g'^1(\overline{n},\overline{j}^1) = g^1(\overline{n},\overline{i}^1) \wedge Q^1(\overline{n},\overline{i}^1)$$

and if, moreover, there exists $\overline{j}^1$ such that $D(\overline{n}) \wedge Q'^1(\overline{n},\overline{j}^1)$ and $g'^1(\overline{n},\overline{j}^1) \ge 1$ then

$$D \vdash \mathsf{L}_{g'^2(\overline{n},\overline{j}^2)}^{Q'^2(\overline{n},\overline{j}^2)}(\ldots \mathsf{L}_{g'^k(\overline{n},\overline{j}^k)}^{Q'^k(\overline{n},\overline{j}^k)}(\alpha)\ldots) \preccurlyeq \mathsf{L}_{g^2(\overline{n},\overline{i}^2)}^{Q^2(\overline{n},\overline{i}^2)}(\ldots \mathsf{L}_{g^k(\overline{n},\overline{i}^k)}^{Q^k(\overline{n},\overline{i}^k)}(\alpha)\ldots).$$

Let $\tau = \mathsf{L}_{f_1}(\ldots \mathsf{L}_{f_r}(\alpha))$ and assume $D \vdash \tau \to \tau'$. To check whether a family $\{g_\imath(\overline{n},\overline{i}_\imath)\}_{Q(\overline{n},\overline{i}_\imath)}$ approximates $f_\imath$, for all $1 \le \imath \le r$ one uses the following lemma.

Lemma 2 (Checking). *Let a family of expressions $[\phi_l](\overline{n},\overline{j}_l)$ with $1 \le l \le t$ approximate multivalued-function symbols $\{\phi_1(\overline{n}),\ldots,\phi_t(\overline{n})\}$ that occur in τ' but not in τ. Let $[\tau]$ and $[\tau']$ be obtained by substituting $g_\imath$ and $[\phi_l]$ for the corresponding function symbols $f_\imath$ and ϕ_l. Then $D \vdash [\tau'] \preccurlyeq [\tau]$ implies that $\{g(\overline{n},\overline{i}_\imath)\}_{Q(\overline{n},\overline{i}_\imath)}$ approximates $f_\imath$, where $1 \le \imath \le r$.*

Proof. By induction on the length of the rewriting chain for an arbitrary $\imath$, fixing some $m \in f_\imath(\overline{n})$.

As an example, checking whether $\{i\}_{0\le i\le nm}$ approximates f_{rel_1} reduces to checking the entailments

$$n = 0 \qquad \vdash 0 =?i \wedge 0 \le ?i \le nm$$
$$n \ge 1,\ 0 \le j' \le m,\ 0 \le j \le (n-1)m \vdash j' + j =?i \wedge 0 \le ?i \le nm$$

The decidability problem of checking whether an indexed family of piecewise polynomials approximates a given multivalued size function is treated similarly to the decidability of type checking for the system annotated with such families directly [SvET08b]. In particular, checking is decidable when function definitions satisfy the syntactical condition from [SvKvE07a] and output approximations are finite families of polynomials. Also, checking is decidable for indexed families of piecewise linear polynomials with indices delimited by linear predicates.

Case Study: Haskell's List Library

As a case study we applied our type inference procedure to Haskell's List Library, in particular to the implementation Hugs version September 2006. For reasons of brevity we will omit the function definitions.

Assumptions

Since for reasons of simplicity we work with a basic language, we need to do some assumptions.

- It must be possible to translate the function into our language. Since our type system is more restrictive, sometimes we cannot define an equivalent function for all cases. Take for instance transpose: $\mathsf{L}_n(\mathsf{L}_m(\alpha)) \to \mathsf{L}_m(\mathsf{L}_n(\alpha))$; Firstly, in our language it must be undefined for the empty list because in that case m is undefined. Secondly, the type system requires that the inner lists have all the same length, which is not the case in the Haskell version.
- We ignore classes of types like Eq and Ord.
- We write the functions uncurrified.

Functions that do not return lists

The following functions do not return lists and thus are not interesting from the size dependency point of view:

head	$: \mathsf{L}_n(\alpha) \to \alpha$
null	$: \mathsf{L}_n(\alpha) \to \mathsf{Bool}$
length	$: \mathsf{L}_n(\alpha) \to \mathsf{Int}$
and, or	$: \mathsf{L}_n(\mathsf{Bool}) \to \mathsf{Bool}$
any, all	$: (\alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \to \mathsf{Bool}$
sum, product	$: \mathsf{L}_{Int}(\alpha) \to \mathsf{Int}$
maximum, minimum	$: \mathsf{L}_{Int}(\alpha) \to \mathsf{Int}$
isPrefix, isSuffix, isInfix	$: \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{Bool}$
elem, notElem	$: \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{Bool}$
!!	$: \mathsf{L}_n(\alpha) \times \mathsf{Int} \to \alpha$

The function !! returns the element from the first argument whose index is the second argument.

Functions with exact size relation

For the following functions, the size of the output can be expressed with a unique polynomial with respect to the size of the inputs. These functions are typable in the type systems developed in [SvKvE07a, TSvE09, SvET08b].

append	$: \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha)$
tail	$: \mathsf{L}_n(\alpha) \to \mathsf{L}_{n-1}(\alpha)$
init	$: \mathsf{L}_n(\alpha) \to \mathsf{L}_{n-1}(\alpha)$
map	$: (\alpha \to \beta) \times \mathsf{L}_n(\alpha) \to \mathsf{L}_n(\beta)$
reverse	$: \mathsf{L}_n(\alpha) \to \mathsf{L}_n(\alpha)$
intersperse	$: \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{2*n \dot{-} 1}(\alpha)$
transpose	$: \mathsf{L}_n(\mathsf{L}_m(\alpha)) \to \mathsf{L}_m(\mathsf{L}_n(\alpha))$
concat	$: \mathsf{L}_n(\mathsf{L}_m(\alpha)) \to \mathsf{L}_{n*m}(\alpha)$
scanl	$: (\alpha \times \beta \to \alpha) \times \alpha \times \mathsf{L}_n(\beta) \to \mathsf{L}_{n+1}(\alpha)$
scanl1	$: (\alpha \times \alpha \to \alpha) \times \mathsf{L}_n(\alpha) \to \mathsf{L}_n(\alpha)$
union	$: \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha)$
unionBy	$: (\alpha \times \alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha)$

$$
\begin{array}{ll}
\mathsf{sort} & : \mathsf{L}_n(\alpha) \to \mathsf{L}_n(\alpha) \\
\mathsf{insert} & : \mathsf{L}_n(\alpha) \to \mathsf{L}_{n+1}(\alpha) \\
\mathsf{intersection} & : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{\min(n,m)}(\alpha) \\
\mathsf{intersectBy} & : (\alpha \times \alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{\min(n,m)}(\alpha)
\end{array}
$$

Functions with indexed family size relation

For these functions definitions the size dependency is not exact and thus they need a family of polynomials to express the different possible sizes of the output.

$$
\begin{array}{ll}
\mathsf{takeWhile} & : (\alpha \to \mathsf{Bool}) \to \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0\le i\le n}}(\alpha) \\
\mathsf{dropWhile} & : (\alpha \to \mathsf{Bool}) \to \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0\le i\le n}}(\alpha) \\
\mathsf{inits},\ \mathsf{tails} & : \mathsf{L}_n(\alpha) \to \mathsf{L}_{n+1}(\mathsf{L}_{\{i\}_{0\le i\le n}}(\alpha)) \\
\mathsf{filter} & : (\alpha \to \mathsf{Bool}) \to \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0\le i\le n}}(\alpha) \\
\mathsf{elemIndices} & : \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0\le i\le n}}(\mathsf{Int}) \\
\mathsf{findIndices} & : (\alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0\le i\le n}}(\mathsf{Int}) \\
\mathsf{nub} & : \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0\le i\le n}}(\alpha) \\
\mathsf{nubBy} & : (\alpha \times \alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0\le i\le n}}(\alpha) \\
\mathsf{delete} & : \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{\max_0(n-i)\}_{0\le i\le 1}}(\alpha) \\
\mathsf{deleteBy} & : (\alpha \times \alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{\max_0(n-i)\}_{0\le i\le 1}}(\alpha) \\
\setminus\setminus & : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{\{\max_0(n-i)\}_{0\le i\le m}}(\alpha) \\
\mathsf{deleteFirstsBy} & : (\alpha \times \alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{\{\max_0(n-i)\}_{0\le i\le m}}(\alpha)
\end{array}
$$

The function $\setminus\setminus$ deletes all the elements of its second argument that are present its first argument.

Functions that use other data types

The language presented in this article has only the basic types Bool, Int and polymorphic lists. However, it is possible to add pairs and algebraic data types following ideas from [TSvE09]. In particular we need sized naturals, Nat, and Maybe. With these additions we would be able to deal with many more functions.

$$
\begin{array}{ll}
\mathsf{mapAccumL},\ \mathsf{mapAccumR} & : (\alpha \to \beta \to (\alpha, \gamma)) \times \alpha \times \mathsf{L}_n(\beta) \to (\alpha, \mathsf{L}_n(\gamma)) \\
\mathsf{span},\ \mathsf{break},\ \mathsf{partition} & : (\alpha \to \mathsf{Bool}) \to \mathsf{L}_n(\alpha) \to (\mathsf{L}_{\{i\}_{0\le i\le n}}(\alpha),\ \mathsf{L}_{\{i\}_{0\le i\le n}}(\alpha))
\end{array}
$$

The type of a span, does not capture the fact that the sum of the length of the resulting lists is equal to the length of the input list. This could be achieved if the index is part of the pair data structure and is shared by its elements. Then we could express

$$\mathsf{span},\ \mathsf{break},\ \mathsf{partition} : (\alpha \to \mathsf{Bool}) \to \mathsf{L}_n(\alpha) \to (\mathsf{L}_{\{i\}}(\alpha),\ \mathsf{L}_{\{n-i\}}(\alpha))_{\{0\le i\le n\}}$$

The following types do not use this extension since it is not covered by our inference procedure. We write $\mathsf{Nat}(n)$ to denote a natural number to which we assign the symbolic value n. This value can be used in size annotation of the output.

$$
\begin{array}{ll}
\mathsf{replicate} & : \mathsf{Nat}(n) \times \alpha \to \mathsf{L}_n(\alpha) \\
\mathsf{take} & : \mathsf{Nat}(n) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{\{i\}_{0 \le i \le n}}(\alpha) \\
\mathsf{drop} & : \mathsf{Nat}(n) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{max_0(m-n)}(\alpha) \\
\mathsf{splitAt} & : \mathsf{Nat}(n) \times \mathsf{L}_m(\alpha) \to (\mathsf{L}_{\{i\}_{0 \le i \le n}}(\alpha),\ \mathsf{L}_{\{max_0(m-i)\}_{0 \le i \le n}}(\alpha)) \\
\mathsf{zip} & : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\beta) \to \mathsf{L}_{max_0(a-max_0(b-a))}((\alpha,\ \beta)) \\
\mathsf{unzip} & : \mathsf{L}_n((\alpha,\ \beta)) \to (L(\alpha, n), L(\beta, n))
\end{array}
$$

We could also specify a more restricted type for zip where we require both lists to have the same length:

$$\mathsf{zip}\colon \mathsf{L}_n(\alpha) \to \mathsf{L}_n(\beta) \to \mathsf{L}_n((\alpha,\ \beta))$$

Functions not covered by our type system

There are functions whose sized types cannot be expressed in our type system. We can divide these in four categories:

- *The size of the result cannot be determined statically.* These functions create a list in a way such that it is not possible to determine its length statically, because it depends on the value of the arguments. In the list library we find unfoldr, whose Haskell type is $(\beta \to \mathsf{Maybe}(\alpha,\ \beta)) \times \beta \to \mathsf{L}(\alpha)$.
- *The size of the output depends on a higher-order parameter.* The only example in the list library is concatMap, whose type could be expressed as $(\alpha \to \mathsf{L}_m(\beta)) \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{n*m}(\beta)$. However, we do not allow such size dependencies in our type system to keep tractability of our type inference procedure.
- *The size of the output is infinite* We work only with finite lists, hence we cannot deal with functions that return infinite lists. If ∞ represents and infinite length, we could say that

$$
\begin{array}{ll}
\mathsf{iterate} & : (\alpha \to \alpha) \times \alpha \to \mathsf{L}_\infty(\alpha) \\
\mathsf{repeat} & : \alpha \to \mathsf{L}_\infty(\alpha) \\
\mathsf{cycle} & : \mathsf{L}_n(\alpha) \to \mathsf{L}_\infty(\alpha)
\end{array}
$$

5 Related Work

This research extends our work [SvKvE07a, vKSvE07, TSvE09] about shapely function definitions that have a single-valued, exact input-output polynomial size functions. Our non-monotonic framework resembles [AAG+07] in which the authors describe *monotonic* resource consumption for JAVA bytecode by means of Cost Equation Systems (CESs), which are similar to, but more general than recurrence equations. CESs express the cost of a program in terms of the size of its input data [AAGP09]. The COSTA system tries to infer a symbolic bound of the program's resource consumption, with respect to a given cost model. In order to obtain a closed form for such recurrence relations, COSTA includes a

dedicated solver called PUBS [AAGP08]. However, the cost functions they consider are limited in non-monotonicity: roughly, non-monotonic sub-expressions must be linear.

Our approach is related to size analysis with polynomial quasi-interpretations [BMM04, Ama05]. There, a program is interpreted as a *monotonic* polynomial extended with the max operation. For instance, Cons(hd, tl) is interpreted as $T + 1$, where T is a numerical variable abstracting tl. Using such interpretations one obtains upper monotonic-polynomial bounds for size functions. The main difference with our approach is that we are interested in non-monotonic lower and upper bounds. In particular, we may infer the size function $(n-m)^2$ for $\mathsf{sqdiff} : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{(n-m)^2}(\alpha)$ (in this simple example the tight lower and upper bounds coincide), see e.g. [vKSvE07]. To our knowledge, *non-monotonic* quasi-interpretations have not been studied for size analysis, but only for proving termination [HM04b]. In this work one considers some unspecified algorithmically decidable classes of non-negative and negative polynomials and introduces abstract variables for the rest.

The EmBounded project was concerned with identifying and certifying resource-bounded code in Hume, a domain-specific high-level programming language for real-time embedded systems. In his thesis, Pedro Vasconcelos [Vas08] uses abstract interpretation to automatically infer linear approximations of the sizes of recursive data types and the stack and heap of recursive functions written in a subset of Hume.

Several papers have studied programming languages with *implicit computational complexity* properties [GMR08, ABT07]. This line of research is motivated both by the perspective of automated complexity analysis and by fundamental goals, in particular to give natural characterisations of complexity classes, like PTIME or PSPACE.

Resource analysis may be performed within a *Proof Carrying Code* framework. In [AM06] the authors introduce resource policies for mobile code to be run on smart devices. Policies are integrated into a proof-carrying code architecture. Two forms of policies are used: *guaranteed policies* which come with proofs and *target policies* which describe limits of the device.

6 Conclusions

This chapter presents a size-aware type system that describes multivalued size functions expressing the dependency between the sizes of inputs and the output size of a function definition. It allows to approximate multivalued output size functions via indexed *non-monotonic* polynomials augmented with the $\max_0$ operation. This feature greatly increases the applicability of our earlier size analysis, which was limited to exact sizes. The extra expressibility comes at a cost: we have crossed the border of decidability. However, this does not make the analysis infeasible in practice.

Conclusions

This chapter draws some brief conclusions about the results presented in this thesis. First we compare the kind of result obtained in each chapter, then we discuss the different formal models that we developed, and finally we outline how the different results can be integrated in new research.

Main Results

In Chapter 2 we proved some concrete properties of a microkernel, namely, two properties that ensure the lack of deadlock under certain circumstances and that all program assertions (that can be expressed in our model) hold in every possible execution. It was while attempting to prove one of these assertions that we uncovered a programming error that could cause the system to fail. Since we did abstractions on both the system and the model of the semantics, the properties that were proved increase our confidence in the microkernel, but do not ensure that it is fully correct. However, the discovery of the bug had concrete implications: the source code was changed to avoid it. But the main result is not the properties that were proven or even the error found, it is the abstraction technique we used to reason about concurrent components: the *Preemption Abstraction*.

In Chapter 3 we described how to translate a security automaton that encodes some desired property into JML annotations that produce an exception if the property is breached. Here the main result is not the proof of some concrete properties of a program like the ones is Chapter 2. It is the translations algorithm, its correctness proof and some of the ideas used during the modelling phase, like parametrised semantics.

In Chapter 4, the last chapter of Part I, we studied the effect an assignment can have on a recursive data structure. In this case the main result is the set of the lemmas that describe how an assignment can change (or not) a path in the heap. These are not properties of a concrete program, but properties of the language. These *meta properties* can in turn be used to prove properties of programs written in that language.

The main results from Chapters 6 and 7 in Part II are the size-aware type systems they introduce. These type systems were proven correct with respect to the operational semantics of the language and its decidability classes were stud-

ied. For the type system of Chapter 6 there is also a prototype implementation available on the web.

Formal Models

All chapters in Part I have used PVS as a tool to formally stablish the desired result. For each of these chapters we have developed a simplified model of the semantics of an imperative language. In every case we strived for simplicity rather than completeness.

By far the most complex is the model presented in Chapter 3. The semantics included an accurate description of method calls and while loops. Side effects, i.e. changes to the state that happen during the evaluation of an expression, were taken into account. We also modelled programs monitored by a security automata, and programs that can either respect or ignore JML annotations. To avoid having many different, yet very similar, program semantics for each kind of program, we devised a semantics that is parametric on the differences.

In the semantics defined in Chapter 4 we concentrated on modelling the heap and assignments in the presence of aliasing, since the intention was to investigate how to describe the changes that an assignment can do to a path in the heap. Method calls and while loops were not modelled in detail, they were only approximated by PVS function calls and recursive functions, respectively. We have traded the ability to deal with side effects for simplicity of reasoning. However, if we reason at the object level rather than at the expression level, there is no need to have side effects into account.

The semantics developed for Chapter 2 is fairly simple. Unlike the other two semantics which do a deep embedding, i.e. the syntax of the language is defined as a data type in PVS, this semantics uses a shallow embedding. Hence, for instance, an if-expression was (manually) translated into a PVS if-expression. Functions with side effects were modelled as two functions: one that returned the result and one that did the change to the state.

A valid question is whether we would have benefited from using the same model in all cases (assuming the language was the same). The obvious benefit would be reuse: once the model is developed (even by other people) we would not have to spend time developing a new model. However, what happens in practice is that any realistic model of a semantics is quite complex. See for instance the models used by tools like KeY [BHS07], Loop [Hui01] and Krakatoa [FM07] for Java programs. One would need to spend a considerable time learning the details of the model. Furthermore the size of the proofs and the time needed to develop them would increase manifold since these semantics include many details that were left out by our abstractions. By concentrating on just what we think it is important and abstracting the rest we can pay for the time spent developing the model of the semantics.

However, care must be taken when selecting a subset of the language to ensure that properties proved will remain true in the full language. Another issue is that

it may not be simple to extend the basic model. For example, when defining the semantics presented in Chapter 3, we first thought about defining a data type for plain JAVA programs and then another data type extending the first one by adding JML constructs. But this is not possible when using inductive data types, thus we defined a data type for programs with JML annotations and a semantics parametrised on the behaviour of annotations. We think that we would have benefited from a framework that allowed one to define a semantics from predefined building blocks. Such modular semantics are being developed [Mos04, LH96, DCB11, MSvE11] but to our knowledge they are not in a state of being practically useful.

For the type systems in Part II we offer correctness proofs, but they are not machine-checked. This could be future work, as discussed next.

Integration

We have presented results in different topics that reflect the areas the authors have worked on. The drawback with this type of presentations is that it may not be clear whether the techniques, frameworks and rules that we have obtained can be combined to obtain new results. We see some opportunities for integration.

The most obvious is applying formal methods and theorem provers, as done in the first part of this thesis, to the type systems developed in the second part. This would be a challenging endeavour, but the potential benefits are encouraging. Firstly, we would be able to obtain a formal, machined-checked, proof of the correctness theorem. This would increase our confidence in the results and ensure that there are no gaps left. Also, depending on the specification language used, an executable prototype could be obtained from the specification. Finally, this would enable us to experiment more easily with the type system by for instance changing a type rule and seeing how it affects the set of typable terms.

The size-aware type systems developed in Part II are suited for functional languages, but there is also research aiming at estimating the resource usage of imperative programs [HJ06, HR09, AAMS10, Atk10, SKvE10]. In imperative programs, including object-oriented ones, one of the most difficult issues for resource analysis is modelling assignments involving possibly aliased variables. In this context, we think it would be interesting to explore whether the rules presented in Chapter 4 can help in such modelling.

Appendix: Soundness Proofs

1 Soundness Proof of the Size-Aware Type System for Algebraic Data Types

Before proving the soundness theorem of Chapter 6, we discuss some semantic notions and prove a few technical lemmas.

We assume *benign sharing* of variables [HJ03]. It means that evaluation of an expression leaves intact the regions of the heap, accessible from the free variables of the continuation. This condition is not typable, but may be approximated statically by some type system, such as uniqueness types [BS93, BS96, dVRM08].

To formalise the notion of benign sharing we introduce a *footprint* function $\mathcal{R} : \mathit{Heap} \times \mathit{ObjHeap} \times \mathit{Val} \longrightarrow \mathcal{P}(\mathit{Loc})$, which computes the set of locations accessible in a given heap h, with a corresponding object heap oh, with $dom(oh) = dom(h)$ from a given value:

$$\mathcal{R}(h,\ oh,\ c) = \emptyset$$
$$\mathcal{R}(h,\ oh,\ \ell) = \begin{cases} \emptyset, & \mathit{if}\ \ell \notin dom(h) \\ \{\ell\} \cup \bigcup_{j=1}^{k} \mathcal{R}(h|_{dom(h)\setminus\{\ell\}},\ oh|_{dom(oh)\setminus\{\ell\}},\ h.\ell.C\_field_j), & \mathit{if}\ \ oh(\ell) = C \end{cases}$$

where $f|_X$ denotes the restriction of a (partial) map f to a set X.

We extend $\mathcal{R}$ to stores by $\mathcal{R}(h,\ oh,\ s) = \bigcup_{x \in dom s} \mathcal{R}(h,\ oh,\ s(x))$. So, the operational-semantics rule with benign sharing looks as follows:

$$\frac{\begin{array}{l} s;\ h;\ oh,\ \mathcal{C} \vdash e_1 \rightsquigarrow v_1;\ h_1;\ oh_1 \\ s[x := v_1];\ h_1;\ oh_1,\ \mathcal{C} \vdash e_2 \rightsquigarrow v;\ h';\ oh' \\ h|_{\mathcal{R}(h,\ oh,\ s|_{TV(e_2)})} = h_1|_{\mathcal{R}(h,\ oh,\ s|_{TV(e_2)})} \\ oh|_{\mathcal{R}(h,\ oh,\ s|_{TV(e_2)})} = oh_1|_{\mathcal{R}(h,\ oh,\ s|_{TV(e_2)})} \end{array}}{s;\ h;\ oh, \mathcal{C} \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow v;\ h';\ oh'}\ \textsc{OSLet}$$

Lemma 1.1 (A program value's footprint is in the heap).
$\mathcal{R}(h,\ oh,\ v) \subseteq dom(h)$.

Proof. The lemma is proved by structural induction on domain of the heap h.

$dom(h) = \emptyset$**:** Then $\mathcal{R}(h,\ oh,\ v) = \emptyset$ is trivially a subset of $dom(h)$.
$dom(h) \neq \emptyset$**:**
$v = \imath$ Then, $\mathcal{R}(h,\ oh,\ v) = \emptyset$, which is trivially a subset of $dom(h)$.
$v = \ell$ **and** $dom(h) = (dom(h) \setminus \{\ell\}) \cup \{\ell\}$**:** From the definition of $\mathcal{R}$ we get $\mathcal{R}(h,\ oh,\ \ell) = \{\ell\} \cup \bigcup_{j=1}^{k} \mathcal{R}(h|_{dom(h)\setminus\{\ell\}},\ oh|_{dom(oh)\setminus\{\ell\}},\ h.l.C\_field_j)$. Applying the induction hypotheses we derive that

$$\mathcal{R}(h|_{dom(h)\setminus\{\ell\}},\ oh|_{dom(oh)\setminus\{\ell\}},\ h.\ell.C\_field_j) \subseteq dom(h|_{dom(h)\setminus\{\ell\}})$$

for all $1 \leq j \leq k$. Hence, $\mathcal{R}(h,\ oh,\ l) \subseteq dom(h)$. □

Lemma 1.2 (Extending a heap does not change the footprints of program values). *If $\ell \notin dom(h)$, $h' = h[\ell.C\_field_1 := v_1,\ ..,\ \ell.C\_field_k := v_k]$ for some $v_1, .., v_k$ and $oh' = oh[\ell := C]$ then for any $v \neq \ell$ one has $\mathcal{R}(h,\ oh,\ v) = \mathcal{R}(h',\ oh',\ v)$.*

Proof. The lemma is proved by induction on the size of the (domain of the) heap h.

$dom(h) = \emptyset$**:** Because $h' = [\ell.C\_field_1 := v_1,\ ..,\ \ell.C\_field_k := v_k]$ and $v \neq \ell$ we have $v \notin dom(h')$. Therefore, $\mathcal{R}(h,\ oh,\ v) = \emptyset = \mathcal{R}(h',\ oh',\ v)$.
$dom(h) \neq \emptyset$**:** We proceed by case distinction on v.
$v = \imath$ then $\mathcal{R}(h,\ oh,\ v) = \emptyset = \mathcal{R}(h',\ oh',\ v)$.
$v = \ell'$**:** Let $\ell' \notin dom(h)$. Then $\mathcal{R}(h,\ oh,\ \ell') = \emptyset$ and $\mathcal{R}(h',\ oh',\ \ell') = \emptyset$ because $\ell' \notin dom(h')$ as well, since $\ell \neq \ell'$ and $dom(h') = dom(h) \cup \ell$.
Let $\ell \in dom(h)$. From the definition of $\mathcal{R}$ we get

$$\mathcal{R}(h,\ oh,\ \ell') = \{\ell'\} \cup \bigcup_{j=1}^{k'} \mathcal{R}(h|_{dom(h)\setminus\{\ell'\}},\ oh|_{dom(oh)\setminus\{\ell'\}},\ h.\ell'.C'\_\mathsf{False}_j).$$

where $oh(\ell') = C'$.
Due to the induction assumption, that $h'(\ell') = h(\ell')$ and that

$$h'|_{dom(h')\setminus\{\ell'\}} = h|_{dom(h)\setminus\{\ell'\}}[h.\ell.C\_field_1 := v_1,\ ..,\ h.\ell.C\_field_k := v_k],$$

we know that

$$\begin{aligned}&\mathcal{R}(h|_{dom(h)\setminus\{\ell'\}},\ oh|_{dom(oh)\setminus\{\ell'\}},\ h.\ell'.C'\_\mathsf{False}_j) = \\ &\mathcal{R}(h'|_{dom(h')\setminus\{\ell'\}},\ oh'|_{dom(oh')\setminus\{\ell'\}},\ h'.\ell'.C'\_\mathsf{False}_j)\end{aligned}$$

for all $1 \leq j \leq k'$. So,

$$\begin{aligned}&\mathcal{R}(h',\ oh',\ \ell') \\ &= \{\ell'\} \cup \textstyle\bigcup_{j=1}^{k'} \mathcal{R}(h'|_{dom(h')\setminus\{\ell'\}},\ oh'|_{dom(oh')\setminus\{\ell'\}},\ h'.\ell'.C'\_\mathsf{False}_j) \\ &= \{\ell'\} \cup \textstyle\bigcup_{j=1}^{k'} \mathcal{R}(h|_{dom(h)\setminus\{\ell'\}},\ oh|_{dom(oh)\setminus\{\ell'\}},\ h.\ell'.C'\_\mathsf{False}_j) \\ &= \mathcal{R}(h,\ oh,\ \ell').\end{aligned}$$

□

Lemma 1.3 (Validity for union of variable sets). *For all stores s and ground contexts Γ the predicate $Valid_{\mathsf{store}}(vars_1 \cup vars_2, \Gamma, s, h;\ oh)$ is true if and only if both $Valid_{\mathsf{store}}(vars_1, \Gamma, s, h;\ oh)$ and $Valid_{\mathsf{store}}(vars_2, \Gamma, s, h;\ oh)$ are true.*

The lemma follows immediately from the definition of a valid store.

Lemma 1.4 (Extending heaps preserves model relations). *For all heaps h and h', if $h'|_{dom(h)} = h$ and $oh'|_{dom(h)} = oh$ then $v \models^{h;\ oh}_{\tau^\bullet} w$ implies $v \models^{h';\ oh'}_{\tau^\bullet} w$.*

Proof.
The lemma is proved by induction on the definition of $\models$.

$v = \imath$**:** In this case $\tau^\bullet = \mathsf{Int}$ and $w = \imath$, hence $v \models^{h';\ oh'}_{\tau^\bullet} w$ by the definition.

$v = \ell$ **and a null-ary constructor:**

$$\ell \models^{h;\ oh}_{T^{\mathsf{c}}(\overline{\tau^\bullet}')} C$$

By the definition we trivially obtain $\ell \models^{h';\ oh'}_{T^{\mathsf{c}}(\overline{\tau^\bullet}')} C$.

$v = \ell$ **and a non null-ary constructor:** In this case $\tau^\bullet = T^{\mathsf{n}^0}(\overline{\tau^\bullet}')$ for some n^0 and

$$\ell \models^{h;\ oh}_{T^{\mathsf{n}^0}(\overline{\tau^\bullet}')} C(w_1, \ldots, w_k)$$

for some w_j, such that

$$\begin{array}{l}
\ell \in dom(h), \quad oh(\ell) = C \\
\mathsf{n}^0 = size_T(C(w_1, \ldots, w_k)) \\
C : \tau_1^\bullet \times \ldots \times \tau_k^\bullet \ \to\ \tau^\bullet \\
h.\ell.C_field_1 \models^{h|_{dom(h)\setminus\{\ell\}};\ oh|_{dom(oh)\setminus\{\ell\}}}_{\tau_1^\bullet} w_1 \\
\ldots \\
h.\ell.C_field_k \models^{h|_{dom(h)\setminus\{\ell\}};\ oh|_{dom(oh)\setminus\{\ell\}}}_{\tau_k^\bullet} w_k
\end{array}$$

We want to apply the induction assumption with heaps $h|_{dom(h)\setminus\{\ell\}}$ and $h'|_{dom(h')\setminus\{\ell\}}$ (as h and h', respectively). The condition of the lemma is satisfied because

$$\begin{aligned}
h'|_{dom(h')\setminus\{\ell\}}|_{dom(h|_{dom(h)\setminus\{\ell\}})} &= h'|_{dom(h')\setminus\{\ell\}}|_{dom(h)\setminus\{\ell\}} \\
&= h'|_{dom(h)\setminus\{\ell\}} \\
&= h|_{dom(h)\setminus\{\ell\}}.
\end{aligned}$$

Thus, we apply the assumption $oh'|_{dom(h)} = oh$, the induction assumption and $h.\ell = h'.\ell$ to obtain

$$\begin{array}{l}
\ell \in dom h', \quad oh'(\ell) = C \\
\mathsf{n}^0 = size_T(C(w_1, \ldots, w_k)) \\
h'.\ell.C_field_1 \models^{h'|_{dom(h')\setminus\{\ell\}};\ oh'|_{dom(oh')\setminus\{\ell\}}}_{\tau_1^\bullet} w_1 \\
\ldots \\
h'.\ell.C_field_k \models^{h'|_{dom(h')\setminus\{\ell\}};\ oh'|_{dom(oh')\setminus\{\ell\}}}_{\tau_k^\bullet} w_k
\end{array}$$

Then, $\ell \models_{\tau^\bullet}^{h',\, oh'} C(w_1, \ldots, w_k)$ by the definition. □

Lemma 1.5 (Values only depend on values at their footprints). *For v, h, w, and $\tau^\bullet$, the relation $v \models_{\tau^\bullet}^{h,\, oh} w$ implies*

$$v \models_{\tau^\bullet}^{h|_{\mathcal{R}(h,\, oh,\, v)},\, oh|_{\mathcal{R}(h,\, oh,\, v)}} w$$

Proof. The lemma is proved by induction on the definition of $\models$.

$v = \imath$**:** then $w = \imath$ and $v \models_{\tau^\bullet}^{h|_{\mathcal{R}(h,\, oh,\, v)},\, oh|_{\mathcal{R}(h,\, oh,\, v)}} \imath$.

$v = \ell$ **and a null-ary constructor:** then $w = C_i$ is a null-ary constructor of $\tau^\bullet = T^{\mathsf{c}_i}(\overline{\tau^\bullet}')$ and

$$v \models_{\tau^\bullet}^{h|_{\mathcal{R}(h,\, oh,\, v)},\, oh|_{\mathcal{R}(h,\, oh,\, v)}} C_i$$

$v = \ell$ **and a non null-ary constructor:** then $\tau^\bullet = T^{\mathsf{n}^0}(\overline{\tau^\bullet}')$ for some $\overline{\tau^\bullet}'$, that $w = C(w_1, \ldots, w_k)$ for some $w_1, \ldots, w_k$

$$\begin{array}{l} \ell \in dom(h), \quad oh(\ell) = C \\ \mathsf{n}^0 = size_T(C(w_1, \ldots, w_k)) \\ C\colon \tau_1^\bullet \times \ldots \times \tau_k^\bullet \to \tau^\bullet \\ h.\ell.C\_field_1 \models_{\tau_1^\bullet}^{h|_{dom(h)\setminus\{\ell\}},\, oh|_{dom(oh)\setminus\{\ell\}}} w_1, \\ \ldots \\ h.\ell.C\_field_k \models_{\tau_k^\bullet}^{h|_{dom(h)\setminus\{\ell\}},\, oh|_{dom(oh)\setminus\{\ell\}}} w_k. \end{array}$$

We apply the induction assumption, with the heap $h|_{dom(h)\setminus\{\ell\}}$:

$$\begin{array}{l} \ell \in dom(h), \quad oh(\ell) = C \\ C\colon \tau_1^\bullet \times \ldots \times \tau_k^\bullet \to \tau^\bullet \\ h.\ell.C\_field_1 \models_{\tau_1^\bullet}^{\left\{ \begin{array}{l} h|_{dom(h)\setminus\{\ell\}}|_{\mathcal{R}(h|_{dom(h)\setminus\{\ell\}},\, oh|_{dom(oh)\setminus\{\ell\}},\, h.\ell.C\_field_1)}, \\ oh|_{dom(oh)\setminus\{\ell\}}|_{\mathcal{R}(h|_{dom(h)\setminus\{\ell\}},\, oh|_{dom(oh)\setminus\{\ell\}},\, h.\ell.C\_field_1)} \end{array} \right\}} w_1, \\ \ldots \\ h.\ell.C\_field_k \models_{\tau_k^\bullet}^{\left\{ \begin{array}{l} h|_{dom(h)\setminus\{\ell\}}|_{\mathcal{R}(h|_{dom(h)\setminus\{\ell\}},\, oh|_{dom(oh)\setminus\{\ell\}},\, h.\ell.C\_field_k)}, \\ oh|_{dom(oh)\setminus\{\ell\}}|_{\mathcal{R}(h|_{dom(h)\setminus\{\ell\}},\, oh|_{dom(oh)\setminus\{\ell\}},\, h.\ell.C\_field_k)} \end{array} \right\}} w_k. \end{array}$$

By Lemma 1.1, $\mathcal{R}(h|_{dom(h)\setminus\{\ell\}},\ oh|_{dom(oh)\setminus\{\ell\}},\ h.\ell.C\_field_j) \subseteq dom(h) \setminus \{\ell\}$, and thus for all $1 \leq j \leq k$,

$$\begin{array}{l} h|_{dom(h)\setminus\{\ell\}}|_{\mathcal{R}(h|_{dom(h)\setminus\{\ell\}},\, oh,\, h.\ell.C\_field_j)} = \\ h|_{\mathcal{R}(h|_{dom(h)\setminus\{\ell\}},\, oh|_{dom(oh)\setminus\{\ell\}},\, h.\ell.C\_field_j)} = \\ h|_{\mathcal{R}(h|_{dom(h)\setminus\{\ell\}},\, oh|_{dom(oh)\setminus\{\ell\}},\, h.\ell.C\_field_j)\setminus\{\ell\}}. \end{array}$$

Due to $\ell \in \mathcal{R}(h,\ oh,\ \ell)$, and Lemma 1.4, with

$$\mathcal{R}(h|_{dom(h)\setminus\{\ell\}},\, oh|_{dom(oh)\setminus\{\ell\}},\, h.\ell.C\_field_j) \setminus \{\ell\} \subseteq \mathcal{R}(h,\ oh,\ \ell) \setminus \{\ell\}$$

we have

$$\begin{array}{l}\ell \in dom(h_{\mathcal{R}(h,\,oh,\,\ell)}), \quad oh(\ell) = C\\ \mathsf{n}^0 = size_T(C(w_1, \ldots, w_k))\\ h|_{\mathcal{R}(h,\,oh,\,\ell)}.\ell.C\_field_1 \models^{h|_{\mathcal{R}(h,\,oh,\,\ell)\setminus\{\ell\}},\ oh|_{\mathcal{R}(h,\,oh,\,\ell)\setminus\{\ell\}}}_{\tau_1^\bullet} w_1,\\ \ldots\\ h|_{\mathcal{R}(h,\,oh,\,\ell)}.\ell.C\_field_k \models^{h|_{\mathcal{R}(h,\,oh,\,\ell)\setminus\{\ell\}},\ oh|_{\mathcal{R}(h,\,oh,\,\ell)\setminus\{\ell\}}}_{\tau_k^\bullet} w_k\end{array}$$

Thus, $\ell \models^{h|_{\mathcal{R}(h,\,oh,\,\ell)},\,oh|_{\mathcal{R}(h,\,oh,\,\ell)}}_{\tau^\bullet} C(w_1, \ldots, w_k)$. □

Lemma 1.6 (Equality of the "meanings" of a program value in two heaps follows from the equality of the footprints).
If $h|_{\mathcal{R}(h,\,oh,\,v)} = h'|_{\mathcal{R}(h,\,oh,\,v)}$ *and* $oh|_{\mathcal{R}(h,\,oh,\,v)} = oh'|_{\mathcal{R}(h,\,oh,\,v)}$ *then* $v \models^{h,\,oh}_{\tau^\bullet} w$ *implies* $v \models^{h',oh'}_{\tau^\bullet} w$.

Proof. Assume $v \models^{h,\,oh}_{\tau^\bullet} w$. By Lemma 1.5 we obtain $v \models^{h|_{\mathcal{R}(h,\,oh,\,v)},\,oh|_{\mathcal{R}(h,\,oh,\,v)}}_{\tau^\bullet} w$. From the assumption of the lemma we get $v \models^{h'|_{\mathcal{R}(h,\,oh,\,v)},\,oh'|_{\mathcal{R}(h,\,oh,\,v)}}_{\tau^\bullet} w$. Now we apply Lemma 1.4, which gives $v \models^{h',oh'}_{\tau^\bullet} w$. □

Lemma 1.7 (Validity of updated store).
Given a typing context Γ, *store* s, *heap* h *with* oh, *value* v, *a set of variables* $vars$ *and a variable* $x \notin vars$, *such that* $x \notin doms$, *we have* $Valid_{\mathsf{store}}(vars, \Gamma, s[x := v], h,\ oh) \iff Valid_{\mathsf{store}}(vars, \Gamma, s, h,\ oh)$.

Proof. The lemma follows from the definition of $Valid_{\mathsf{store}}$. □

Lemma 1.8 (Subset of subset of variables).
Given typing context Γ, *stack* s, *a heap* h *with* oh *and set of variables* $vars_1$ *and* $vars_2$ *such that* $vars_2 \subseteq vars_1$, *we have that* $Valid_{\mathsf{store}}(vars_1, \Gamma, s, h,\ oh) \implies Valid_{\mathsf{store}}(vars_2, \Gamma, s, h,\ oh)$.

Proof. The lemma follows from the definition of $Valid_{\mathsf{store}}$. □

The soundness theorem is a partial case of the following lemma.

Theorem 1 (Soundness).
For any s, h, oh, $\mathcal{C}$, e, v, h', oh' *a set of equations* D, *a context* Γ, *a signature* Σ, *and a type* τ, *any size valuation* ϵ, *a type instantiation* η *such that*

- $s;\ h;\ oh,\ \mathcal{C} \vdash e \rightsquigarrow v;\ h';\ oh'$,
- $D;\ \Gamma \vdash_\Sigma e\colon \tau$ *is derivable in the type system, and is a node in some derivation tree, where all functions called in* e *are declared via* letfun,
- D *holds on size variables valuated by* ϵ *(i.e.* D_ϵ *holds)*

if the store is meaningful w.r.t. the context $\eta(\epsilon(\Gamma))$ *then the output value is meaningful w.r.t the type* $\eta(\epsilon(\tau))$.

Proof. For the sake of convenience we will denote $\eta(\epsilon(\tau))$ via $\tau_{\eta\epsilon}$ and $\eta(\epsilon(\Gamma))$ via $\Gamma_{\eta\epsilon}$.

We prove the statement by induction on the height of the derivation tree for the operational semantics. Given $s;\ h;\ oh,\ \mathcal{C} \vdash e \rightsquigarrow v;\ h';\ oh$, we fix some Γ, Σ, and τ, such that $D;\ \Gamma \vdash_{\Sigma} e : \tau$. We fix a valuation $\epsilon \in TV(\Gamma) \cup TV(\tau) \to \mathcal{Z}$, a type instantiation $\epsilon \in TV(\Gamma) \cup TV(\tau) \to \tau^{\bullet}$, such that the assumptions of the lemma hold. We have to show that $Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h',\ oh')$ holds.

OSICons: In this case $v = \imath$ for some constant $c\imath$ and $\tau = \mathsf{Int}$. Then, by the definition we have $\imath \models^{h,\ oh}_{\mathsf{Int}} \imath$ and $Valid_{\mathsf{val}}(v, \mathsf{Int}, h' = h,\ oh' = oh)$.

OSVar: From D_ϵ it follows that $\tau_{\eta\epsilon} = \tau'_{\eta\epsilon}$. From this and $Valid_{\mathsf{store}}(TV(x), (\Gamma \cup x : \tau')_{\eta\epsilon}, h,\ oh, s)$ it follows

$$Valid_{\mathsf{val}}(s(x), \tau_{\eta\epsilon}, h' = h,\ oh' = oh).$$

OSCons-0: in this case $e = C_i$, where C_i is a null-ary constructor of some type T, $v = \ell \notin dom(h)$. From the type derivation we have that $\tau = T^{\mathsf{p}}(\overline{\tau}')$ for some $\overline{\tau}'$, and, moreover, $D \vdash p = \mathsf{c}_i$. By the definition of $\models$ relation we have $\ell \models^{h[\ell.C_i\_field_1 := i],\ oh[\ell := C_i]}_{T^{\mathsf{c}_i}(\overline{\tau}'_{\eta\epsilon})} C_i$, and therefore

$$D_\epsilon \vdash\ \ell \models^{h[\ell.C_i\_field_1 := i],\ oh[\ell := C_i]}_{T^{\mathsf{p}_\epsilon}(\overline{\tau}'_{\eta\epsilon})} C_i.$$

Thus, $Valid_{\mathsf{val}}(v, T^{\mathsf{p}_\epsilon}(\overline{\tau}')_{\eta\epsilon}, h' = h[\ell.C_i\_field_1 := i],\ oh' = oh[\ell := C_i])$.

OSCons: In this case $e = C(x_1, \ldots, x_k)$, $v = \ell \notin dom(h)$. From the typing rule we have that $\tau = T^{\mathsf{p}}(\overline{\tau}')$ for some $\overline{\tau}'$ and there exist τ_j, such that $x_j : \tau_j \subseteq \Gamma$, and one has the instance of C of type $\tau_1 \times \ldots \times \tau_k \to T^{\mathsf{p}}(\overline{\tau}')$. Moreover, $\tau_j = T_j^{\mathsf{p}_j}(*)$ for some p_j if τ_j takes part in the counting sizes, (otherwise think that it is equal to zero).
Since $Valid_{\mathsf{store}}(TV(e), \Gamma_{\eta\epsilon}, s, h,\ oh)$ there exist w_j such that $s(x_j) \models^{h,\ oh}_{\tau_j\ \eta\epsilon} w_j$. From the operational-semantics judgement we have $h' = h[\ell.C\_field_1 := s(x_1), \ldots, \ell.C\_field_k := s(x_k)]$. Therefore, $h'.\ell.C\_field_j \models^{h,\ oh}_{\tau_j\ \eta\epsilon} w_j$. It is easy to see that $h = h'|_{dom h' \setminus \{\ell\}}$ and similarly for oh'. Thus,

$$h'.\ell.C\_field_j \models^{h'|_{dom h' \setminus \{\ell\}};\ oh'|_{dom oh' \setminus \{\ell\}}}_{\tau_j\ \eta\epsilon} w_j.$$

From the typing rule we have that D implies $\mathsf{p} = \mathsf{c} + \sum_{j=1}^{k} a_j \mathsf{p}_j$. For the ground valuation we have $\mathsf{p}_\epsilon = \mathsf{c} + \sum_{j=1}^{k} a_j \mathsf{p}_{j_\epsilon}$. From the definition of the $\models$ relation, it follows that $\mathsf{p}_{j_\epsilon} = size_{T_j}(w_j)$. Therefore, we have that $\mathsf{p}_\epsilon = \mathsf{c} + \sum_{j=1}^{k} a_j size_{T_j}(w_j) = size_T(C(w_1, \ldots, w_k))$. This gives $\ell \models^{h',oh'}_{T^{\mathsf{p}}(\overline{\tau}')_{\eta\epsilon}} C(w_1, \ldots, w_k)$ and thus $Valid_{\mathsf{val}}(\ell, \tau_{\eta\epsilon}, h', oh')$.

OSIfFalse: In this case $e = \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ for some e_1, e_2, and x. Knowing that $D;\ \Gamma \vdash_{\Sigma} e_2 : \tau$, we apply the induction hypothesis to the derivation of $s;\ h;\ oh,\ \mathcal{C} \vdash e_2 \rightsquigarrow v;\ h';\ oh'$ to obtain

$$Valid_{\mathsf{store}}(TV(e_2), \Gamma_{\eta\epsilon}, s, h,\ oh) \implies Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h',\ oh').$$

From $TV(e_2) \subseteq TV(e)$, $Valid_{\mathsf{store}}(TV(e), \Gamma_{\eta\epsilon}, s, h, oh)$, and Lemma 1.8 it follows that $Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h', oh')$.

OSIfTrue: In this case $e = \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ for some e_1, e_2, and x. Knowing that $D;\ \Gamma \vdash_\Sigma e_1 : \tau$, we apply the induction hypothesis to the derivation of $s;\ h;\ oh,\ \mathcal{C} \vdash e_1 \rightsquigarrow v;\ h';\ oh'$ to obtain

$$Valid_{\mathsf{store}}(TV(e_1), \Gamma_{\eta\epsilon}, s, h, oh) \implies Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h', oh')$$

From $TV(e_1) \subseteq TV(e)$, $Valid_{\mathsf{store}}(TV(e), \Gamma_{\eta\epsilon}, s, h, oh)$, and Lemma 1.8 it follows that $Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h', oh')$.

OSLetFun: The result follows from the induction hypothesis for

$$s;\ h;\ oh,\ \mathcal{C}[f := (\overline{x} \times e_1)] \vdash e_2 \rightsquigarrow v;\ h';\ oh',$$

with $D;\ \Gamma \vdash_\Sigma e_2 : \tau$ and the same η, ϵ.

OSLet: In this case $e = \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ for some x, e_1, and e_2 and we have $s;\ h;\ oh,\ \mathcal{C} \vdash e_1 \rightsquigarrow v_1;\ h_1;\ oh_1$ and $s[x := v_1];\ h_1;\ oh,\ \mathcal{C} \vdash e_2 \rightsquigarrow v;\ h';\ oh$ for some v_1 and h_1. We know that $D;\ \Gamma \vdash_\Sigma e_1 : \tau'$, $x \notin \Gamma$ and $D;\ \Gamma, x : \tau' \vdash_\Sigma e_2 : \tau$ for some τ'. Applying the induction hypothesis to the first branch gives $Valid_{\mathsf{store}}(TV(e_1), \Gamma_{\eta\epsilon}, s, h, oh) \implies Valid_{\mathsf{val}}(v_1, \tau'_{\eta\epsilon}, h_1, oh_1)$. Since

$$TV(e_1) \subseteq TV(e_1) \cup (TV(e_2) \setminus \{x\}) = TV(e) \text{ and}$$
$$Valid_{\mathsf{store}}(TV(e), \Gamma_{\eta\epsilon}, s, h, oh),$$

we have from Lemma 1.8 that $Valid_{\mathsf{store}}(TV(e_1), \Gamma_{\eta\epsilon}, s, h, oh)$ holds and thus we have $Valid_{\mathsf{val}}(v_1, \tau'_{\eta\epsilon}, h_1, oh_1)$.
Now apply the induction hypothesis to the second branch to get

$$Valid_{\mathsf{store}}(TV(e_2), \Gamma_{\eta\epsilon} \cup \{x : \tau'\}_{\eta\epsilon}, s[x := v_1], h_1, oh_1) \implies Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h', oh').$$

Fix some $y \in TV(e_2)$. If $y = x$, then

$$Valid_{\mathsf{val}}(v_1, \tau'_{\eta\epsilon}, h_1, oh_1) \implies Valid_{\mathsf{val}}(s[x := v_1](x), \tau'_{\eta\epsilon}, h_1, oh_1).$$

If $y \neq x$, then $s[x := v_1](y) = s(y)$. Because we know that sharing is benign, $h|_{\mathcal{R}(h,\, oh,\, s(y))} = h_1|_{\mathcal{R}(h,\, oh,\, s(y))}$, applying Lemma 1.6 and then Lemma 1.8 we have that

$$s(y) \models^{h,\, oh}_{\Gamma_{\eta\epsilon}(y)} w_y \implies s(y) \models^{h_1,\, oh_1}_{\Gamma_{\eta\epsilon}(y)} w_y \implies s[x := v_1](y) \models^{h_1,\, oh_1}_{\Gamma_{\eta\epsilon}(y)} w_y$$

and thus $Valid_{\mathsf{val}}(s[x := v_1](y), \Gamma_{\eta\epsilon}(y), h_1, oh_1)$.
Hence, $Valid_{\mathsf{store}}(TV(e_2), \Gamma_{\eta\epsilon} \cup \{x : \tau'\}_{\eta\epsilon}, s[x := v_1], h_1, oh_1)$, and therefore, $Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h', oh')$.

OSMatch-0-ary: In this case the expression e has the form

$$e = \texttt{match}\ x\ \texttt{with} \begin{array}{l} \mid C_1(x_{11}, \ldots, x_{1k_1}) \Rightarrow e_1\,. \\ \vdots \\ \mid C_r(x_{r1}, \ldots, x_{rk_r}) \Rightarrow e_r \end{array}$$

The typing context has the form $\Gamma = \Gamma' \cup \{x\colon T^{\mathsf{p}}(\overline{\tau}')\}$ for some Γ', τ', p. The operational-semantics derivation gives $s(x) = \ell$, and C_i is a null-ary constructor of x-s type. Hence

$$s(x) \models^{h,\,oh}_{T^{\mathsf{p}}(\overline{\tau}')_{\eta\epsilon}} C_i$$

gives $\epsilon(\mathsf{p}) = \mathsf{c}_i$. From the typing derivation for $D;\ \Gamma \vdash_\Sigma e\colon \tau$ we know that $\mathsf{p} = \mathsf{c}_i,\ D;\ \Gamma' \vdash_\Sigma e_i\colon \tau$. Applying the induction hypothesis, with $D_\epsilon \wedge \mathsf{p}_\epsilon = \mathsf{c}_i$ then yields $\mathit{Valid}_{\mathsf{store}}(TV(e_i), \Gamma'_{\eta\epsilon}, s, h,\ oh) \implies \mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h', oh')$. From $TV(e_i) \subseteq TV(e),\ \mathit{Valid}_{\mathsf{store}}(TV(e), \Gamma_{\eta\epsilon}, s, h,\ oh)$ it follows that

$$\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h',\ oh').$$

OSMatch-C$_\mathbf{i}$: In this case, again, the expression e has the form

$$e = \texttt{match}\ x\ \texttt{with} \begin{array}{l} \mid C_1(x_{11}, \ldots, x_{1k_1}) \Rightarrow e_1\,. \\ \vdots \\ \mid C_r(x_{r1}, \ldots, x_{rk_r}) \Rightarrow e_r \end{array}$$

The operational-semantics derivation gives $oh(s(x)) = C_i$. The typing context has the form $\Gamma = \Gamma' \cup \{x\colon T^{\mathsf{p}}(\overline{\tau'})\}$ for some Γ', $\overline{\tau'}$, p. Hence validity

$$s(x) \models^{h,\,oh}_{T^{\mathsf{p}}(\overline{\tau'})_{\eta\epsilon}} C_i(w_1, \ldots, w_k)$$

gives $h.s(x).C_i\_field_j \models^{h,\,oh}_{\tau_j\,\eta\epsilon} w_j$.
From the typing derivation for $D;\ \Gamma \vdash_\Sigma e\colon \tau$ we know that

$$D,\ \mathsf{p} = \mathsf{c}_i + \sum_{j=1}^{k_i} a_{ij}\mathsf{n}_{ij},\ \Gamma',\ x\colon T^{\mathsf{p}}(\overline{\tau'}),\ x_{ij}\colon T^{n_{ij}} \vdash_\Sigma e_i\colon \tau.$$

To apply the induction hypothesis we must extend the valuation ϵ to n_{ij} (call this extension ϵ') so that

$$D_{\epsilon'} \wedge \mathsf{p}_{\epsilon'} = \mathsf{c}_i + \sum_{j=1}^{k_i} a_{ij}\mathsf{n}_{ij\,\epsilon'} \text{ holds.}$$

We assign $n_{ij} = \mathit{size}_{T_{ij}}(w_j)$, taken from the definition of $\models$-relation for $h.s(x).C_i\_field_j$. Then from the definition of $\models$-relation for $s(x)$ it follows that $p_{\epsilon'} = p_\epsilon = \mathit{size}_T(C(w_1, \ldots, w_k)) = \mathsf{c}_i + \sum_{j=1}^{k_i} a_{ij}\mathit{size}_{T_{ij}}(w_j) = \mathsf{c}_i + \sum_{j=1}^{k_i} a_{ij}\mathsf{n}_{ij\,\epsilon'}$.

Applying the induction hypothesis, with $D \wedge \mathsf{p} = \mathsf{c}_i + \sum_{j=1}^{k_i} a_{ij}\mathsf{n}_{ij}$ with ϵ', η yields

$$\mathit{Valid}_{\mathbf{store}}(TV(e_i), (\Gamma', x\colon T^{\mathsf{p}}(\overline{\tau'}), x_{ij}\colon T^{n_{ij}})_{\eta\epsilon'}), s[\ldots, x_{ij} := h.s(x).C_i\_\mathit{field}_j, \ldots], h,\ oh) \implies \mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon'}, h', oh').$$

Then we have to show the antecedent of this implication. It is easy to see, that $FVe_i) \subseteq TV(e) \cup \{x_{i1}, \ldots, x_{ik_i}\}$. We trivially have that

$$\mathit{Valid}_{\mathbf{store}}(TV(e), (\Gamma'\ x\colon T^{\mathsf{p}}(\overline{\tau'}))_{\eta\epsilon'}, s, h,\ oh).$$

Further, from the model relations above we have that

$$\mathit{Valid}_{\mathsf{val}}(s[\ldots,\ x_{ij} := h.s(x).C_i\_\mathit{field}_j,\ \ldots](x_{ij}), \tau_{j\,\eta\epsilon'}, h, oh).$$

So, the store for evaluation of e_i is meaningful as well.
We apply the induction hypothesis and get $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon'}, h', oh')$ for the valuation ϵ'. The last step is to show that $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h', oh')$ for the initial valuation ϵ. This is trivially true, because τ has only free size variables from $dom(\epsilon)$, where ϵ and ϵ' coincide.

OSFun: We want to apply the induction assumption to

$$[y_1 := v_1, \ldots, y_k := v_k];\ h;\ oh,\ \mathcal{C} \vdash e_f \rightsquigarrow v;\ h';\ oh'.$$

Since the original typing judgement is a node in a derivation tree, where all called in e functions are defined via letfun, there must be a node in the derivation tree with $\mathsf{True},\ y_1 : \tau_1^\circ, \ldots, y_k : \tau_k^\circ \vdash_\Sigma e_f\colon \tau'$.
We take η' and ϵ', such that

- $\eta'(\alpha) = \tau_{\alpha\eta\epsilon}$, where τ_α is such that α is replaced by τ_α in the instantiation of the signature in *this* application of the FunApp-rule.
- $\epsilon'(n_{ij}) = p_{ij_\epsilon}$, where n_{ij} is replaced by p_{ij} in the instantiation of the signature in *this* application of the FunApp-rule.

True ("no conditions") holds trivially on ϵ'.
From the induction assumption we have

$$\mathit{Valid}_{\mathbf{store}}((y_1, \ldots, y_k), (y_1 : \tau_{1\,\eta'\epsilon'}^\circ, \ldots, y_k : \tau_{k\,\eta'\epsilon'}^\circ), [y_1 := v_1, \ldots, y_n := v_n], h;\ oh) \implies \mathit{Valid}_{\mathsf{val}}(v, \tau'_{\eta'\epsilon'}, h';\ oh').$$

From $\mathit{Valid}_{\mathbf{store}}(TV(e), \Gamma_{\eta\epsilon}, s, h;\ oh)$ we have validity of the values of the actual parameters: $v_j \models^{h;\,oh}_{\Gamma_{\eta\epsilon}(x_j)} w_j$ for some w_j, where $1 \le j \le k$. Since $\Gamma_{\eta\epsilon}(x_j) = \tau^\circ_{j\,\eta'\epsilon'}$, the left-hand side of the implication holds, and one obtains $\mathit{Valid}_{\mathsf{val}}(v, \tau'_{\eta'\epsilon'}, h';\ oh')$.
Since D_ϵ implies $\tau'[\ldots\alpha := \tau_\alpha\ldots][\ldots n_{ij} := p_{ij}\ldots]_{\eta\epsilon} = \tau_{\eta\epsilon}$, and by the definition of η', ϵ' we have $\tau'_{\eta'\epsilon'} = \tau'[\ldots\alpha := \tau_{\alpha\eta\epsilon}\ldots][\ldots n_{ij} := p_{ij_\epsilon}\ldots]$ one easily obtains $\tau_{\eta\epsilon} = \tau'_{\eta'\epsilon'}$ and, eventually, $\mathit{Valid}_{\mathsf{val}}(v, \tau'_{\eta\epsilon}, h';\ oh')$.

□

2 Soundness Proof of the Type System for Collected Size Semantics

Next we give the full proof of the soundness theorem of Chapter 7. As in Section 1 of this appendix, we assume *benign sharing* of variables. We will use a similar formalisation via the footprint function $\mathcal{R}(h, v)$, which has the same properties proved for benign sharing in the previous section.

We recall the lemma relating the length of a lists to the amount of memory actually allocated.

Lemma 2.1 (Consistency of model relation). *The relation* $\mathtt{adr} \models^{h}_{\mathsf{L}_s(\tau^\bullet)} w$ *implies that* $length_h(\mathtt{adr}) \in s$.

It is simple to prove that valuations and instantiations distribute over types and size functions in the following way: $\eta\epsilon((\mathsf{L}_{f(\overline{n})}(\tau))) = \mathsf{L}_{f(\epsilon(\overline{n}))}(\eta(\epsilon(\tau)))$. For the sake of convenience we abbreviate $D(\epsilon(\overline{n}))$ to D_ϵ, $\eta(\epsilon(\tau))$ to $\tau_{\eta\epsilon}$ and $\eta(\epsilon(\Gamma))$ to $\Gamma_{\eta\epsilon}$.

The next lemma says that rewriting implies set-theoretic inclusion of types.

Lemma 2.2 (Rewriting preserves model relation,). *Let* $D(\overline{n}) \vdash \tau \to \tau'$. *Let a valuation* ϵ *and a type instantiation* η *be such that* $v \models^{h}_{\tau'_{\eta\epsilon}} w$ *and* D_ϵ *hold. Then* $v \models^{h}_{\tau_{\eta\epsilon}} w$ *holds as well.*

Proof. By induction on $\models$. Let $\epsilon(\overline{n}) = \overline{n}_0$.

The case where v is an integer or a boolean is straightforward since τ' and τ will be Int or Bool, respectively. Assume $v = \mathtt{NULL}$. Then $0 \in s'(\overline{n}_0)$ and $w = \mathtt{[]}$. Since $s(\overline{n}_0) \to s'(\overline{n}_0)$, that is $s'(\overline{n}_0) \subseteq s(\overline{n}_0)$, we have $0 \in s(\overline{n}_0)$ and $v \models^{h}_{\tau_{\eta\epsilon}} \mathtt{[]}$.

Now let $v = \ell$ and $w = w_{hd} :: w_{tl}$ where $h.\ell.hd \models^{h|_{dom(h)\setminus\{\ell\}}}_{\tau'''_{\eta\epsilon}} w_{hd}$ and $h.\ell.tl \models^{h|_{dom(h)\setminus\{\ell\}}}_{\mathsf{L}_{s'(\overline{n}_0)-1}(\tau'''_{\eta\epsilon})} w_{tl}$. Also let $\tau = \mathsf{L}_{s(\overline{n})}(\tau'')$ and $\tau' = \mathsf{L}_{s(\overline{n})}(\tau''')$ for some τ'', τ'''. Since there is $n \in s(\overline{n}_0)$, $n \geq 1$ we have $D \vdash \tau'' \to \tau'''$ and by induction $h.\ell.hd \models^{h|_{dom(h)\setminus\{\ell\}}}_{\tau''_{\eta\epsilon}} w_{hd}$. Since $s(\overline{n}) \to s'(\overline{n})$ we have $s(\overline{n}) - 1 \to s'(\overline{n}) - 1$ and by induction $h.\ell.tl \models^{h|_{dom(h)\setminus\{\ell\}}}_{\mathsf{L}_{s(\overline{n}_0)-1}(\tau''_{\eta\epsilon})} w_{tl}$. □

This lemma may seem counterintuitive on a first sight because it looks like a type preservation lemma where the type τ and τ' are swapped. However, a rewriting rule is different from an evaluation step. The idea behind this lemma is that on a rewriting rule there are several choices on the left hand side (τ) and one in particular is chosen to obtain the right hand side (τ'). So if a value has type τ', it also has type τ.

Theorem 1 (Soundness). *For any store* s, *heaps* h *and* h', *closure* $\mathcal{C}$, *expression* e, *value* v, *context* Γ, *quantifier-free formula* D, *signature* Σ, *type* τ, *size valuation* ϵ, *and type instantiation* η *such that*

- *the expression* e *terminates with the value* v, *i.e. in terms of operational semantics the relation* $s; h; \mathcal{C} \vdash e \rightsquigarrow v; h'$ *holds,*

- $D, \Gamma \vdash_\Sigma e\colon \tau$ *is a node in the derivation tree for some function body,*
- $dom(s) = dom(\Gamma)$,
- $D(\epsilon(\overline{n}))$ *holds, where* $\overline{n}$ *is the set of size variables from* $dom(\Gamma \cup D)$,
- $Valid_{\mathsf{store}}(dom(s), \eta(\epsilon(\Gamma)), s, h)$ *holds,*

then the return value v *is valid according to its return type* τ*, i.e.*

$$Valid_{\mathsf{val}}(v, \eta(\epsilon(\tau)), h')$$

holds.

Proof. The proof is done by induction on the size of the derivation tree for the operational-semantic judgement.

To avoid clash of notations, in this proof we use $f(\overline{n})$ instead of $s(\overline{n})$ to denote a size annotation with free size variables $\overline{n}$ (here s will be used for the store of the operational semantics).

One can easily check by induction that $TV(\tau) \subseteq TV(\Gamma)$. Fix a valuation $\epsilon\colon SV(\Gamma) \cup SV(D) \to \mathcal{R}$, and a type instantiation $\eta\colon TV(\Gamma) \to \tau^\bullet$ such that the assumptions of the lemma hold. We must show that $Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$ holds.

OSNull: In this case $v = \texttt{NULL}$, and according to the definition of the model relation we have $\texttt{NULL} \models^h_{\mathsf{L}_0(\tau'_{\eta\epsilon})} \texttt{[]}$ for τ' from the typing rule. Now we use the fact that $D \vdash \tau \to \mathsf{L}_0(\tau')$ and D_ϵ holds to obtain by Lemma 2.2 that $\texttt{NULL} \models^h_{\tau_{\eta,\epsilon}} \texttt{[]}$.

OSVar: In this case $v = s(\mathsf{z})$. From $Valid_{\mathsf{store}}(dom(s), (\Gamma' \cup (x : \tau')_{\eta\epsilon}, h, s)$ for the corresponding τ' it follows that $s(\mathsf{z}) \models^h_{\tau'} w$ for some w. Now, by Lemma 2.2, $D \vdash \tau \to \tau'$ and D_ϵ imply $v \models^h_{\tau_{\eta,\epsilon}} w$.

OSCons: In this case $e = \mathsf{Cons}(\mathsf{hd}, \mathsf{tl})$ and Γ is "Γ', $\mathsf{hd} : \tau_1, \mathsf{tl} : \mathsf{L}_{f(\overline{n})}(\tau_2)$" for some Γ', hd, tl, f and τ'. Since $Valid_{\mathsf{store}}(dom(s), \Gamma_{\eta\epsilon}, s, h)$ there exist w_{hd} and w_{tl} such that $s(\mathsf{hd}) \models^h_{\tau_{1\,\eta\epsilon}} w_{hd}$ and $s(\mathsf{tl}) \models^h_{\mathsf{L}_{f(\epsilon(\overline{n}))}(\tau_{2\eta\epsilon})} w_{tl}$. From the operational semantics judgement we have that $v = \ell$ for some location $\ell \notin dom(h)$, and $h' = h[\ell.hd := s(\mathsf{hd}),\ \ell.tl := s(\mathsf{tl})]$. Therefore, $h'.\ell.hd \models^h_{\tau_{1\,\eta\epsilon}} w_{hd}$ and $h'.\ell.tl \models^h_{\mathsf{L}_{f(\epsilon(\overline{n}))}(\tau_{2\eta\epsilon})} w_{tl}$ also hold. It is easy to see that $h = h'|_{dom(h')\setminus\{\ell\}}$. Thus,

$$h'.\ell.hd \models^{h'|_{dom(h')\setminus\{\ell\}}}_{\tau_{1\,\eta\epsilon}} w_{hd}$$
$$h'.\ell.tl \models^{h'|_{dom(h')\setminus\{\ell\}}}_{\mathsf{L}_{f(\epsilon(\overline{n}))}(\tau_{2\eta\epsilon})} w_{tl}$$

Applying Lemma 2.2 to $D \vdash \tau_2 \to \tau_1$ and D_ϵ we obtain $\ell \models^{h'}_{\mathsf{L}_{f(\epsilon(\overline{n}))+1}(\tau_{2\eta\epsilon})} w_{hd} :: w_{tl}$. Using the fact that $D \vdash \tau \to \mathsf{L}_{f(\overline{n})+1}(\tau'_2)$ and the same lemma we obtain $v \models^h_{\tau_{\eta\epsilon}} w_{hd} :: w_{tl}$.

OSIfTrue: In this case $e = \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ for some e_1, e_2, and x. We apply the induction hypothesis to the derivation of $s;\ h;\ \mathcal{C} \vdash e_1 \rightsquigarrow v;\ h'$, with the same η, ϵ to obtain $Valid_{\mathsf{store}}(dom(s), \Gamma_{\eta\epsilon}, s, h) \implies Valid_{\mathsf{val}}(v, \tau_{1\ \eta\epsilon}, h')$.

Due to the condition of the lemma, we have $Valid_{\mathsf{val}}(v, \tau_{1\eta\epsilon}, h')$. Using Lemma 2.2 and the fact that $D \vdash \tau \to \tau_1$ holds (due to the definition of $D \vdash \tau \to \tau_1|\tau_2$), we obtain $Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$.

OSIfFalse: Exactly as the true-case, but with e_2 instead of e_1.

OSLetFun: The result follows from the induction hypothesis for

$$s;\ h;\ \mathcal{C}[\mathsf{f} := (\mathsf{z} \times e_1)] \vdash e_2 \rightsquigarrow v;\ h',$$

with $\Gamma \vdash_\Sigma e_2 : \tau$ and the same η, ϵ.

OSLet: In this case e is $\mathsf{let}\ \mathsf{z} = e_1\ \mathsf{in}\ e_2$ for some z, e_1, and e_2 and we have $s;\ h;\ \mathcal{C} \vdash e_1 \rightsquigarrow v_1;\ h_1$ and $s[\mathsf{z} := v_1];\ h_1;\ \mathcal{C} \vdash e_2 \rightsquigarrow v;\ h'$ for some v_1 and h_1. Applying the induction hypothesis to the let-binding branch in let-rule's antecedent gives $Valid_{\mathsf{store}}(dom(s), D, \Gamma_{\eta\epsilon}, s, h) \implies Valid_{\mathsf{val}}(v_1, \tau'_{\eta\epsilon}, h_1)$. Now apply the induction hypothesis to the let-body branch to get

$Valid_{\mathsf{store}}(dom(s[\mathsf{z} := v_1]), \Gamma_{\eta\epsilon} \cup \{\mathsf{z}\colon \tau'_{\eta\epsilon}\}, s[\mathsf{z} := v_1], h_1) \implies Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$

Fix some $\mathsf{z}' \in dom(s[\mathsf{z} := v_1])$. If $\mathsf{z}' = \mathsf{z}$, then $Valid_{\mathsf{val}}(v_1, \tau'_{\eta\epsilon}, h_1)$ implies $Valid_{\mathsf{val}}(s[\mathsf{z} := v_1](\mathsf{z}), \tau'_{\eta\epsilon}, h_1)$. If $\mathsf{z}' \neq \mathsf{z}$, then $s[\mathsf{z} := v_1](\mathsf{z}') = s(\mathsf{z}')$. Sharing of data structures in the heap is benign (no destructive pattern matching and assignments), hence $h|_{\mathcal{R}(h, s(\mathsf{z}'))} = h_1|_{\mathcal{R}(h, s(\mathsf{z}'))}$. Thus, we have that $s(\mathsf{z}') \models^h_{\Gamma_{\eta\epsilon}(\mathsf{z}')} w'_z$ implies $s(\mathsf{z}') \models^{h_1}_{\Gamma_{\eta\epsilon}(\mathsf{z}')} w'_z$ implies $s[\mathsf{z} := v_1](\mathsf{z}') \models^{h_1}_{\Gamma_{\eta\epsilon}(\mathsf{z})} w_{\mathsf{z}'}$. So, $Valid_{\mathsf{val}}(s[\mathsf{z} := v_1](\mathsf{z}'), \Gamma_{\eta\epsilon}(\mathsf{z}'), h_1)$. Hence,

$Valid_{\mathsf{store}}(dom(s[\mathsf{z} := v_1]), \Gamma_{\eta\epsilon} \cup \{\mathsf{z}\colon \tau'_{\eta\epsilon}\}, s[\mathsf{z} := v_1], h_1)$

and we can apply the induction assumption to the let-body.

OSMatch-Nil: In this case $e = \mathsf{match\ l\ with\ |\ Nil} \Rightarrow e_1 \mid \mathsf{Cons(hd,\ tl)} \Rightarrow e_2$ for some l, hd, tl, e_1, and e_2. The typing context has the form $\Gamma = \Gamma' \cup \{\mathsf{l} : \mathsf{L}_{f(\overline{n})}(\tau)\}$ for some Γ', τ' and f. The operational-semantics derivation gives $s(\mathsf{l}) = \mathtt{NULL}$, hence, validity for $s(\mathsf{l})$ gives $0 \in f(\epsilon(\overline{n}))$. Therefore, $s(\mathsf{l}) \models^h_{\mathsf{L}_{f_\epsilon(\overline{n})}(\tau')} []$. This means that $Valid_{\mathsf{store}}(dom(s), (\Gamma'_{\eta\epsilon'}, \mathsf{l}\colon \mathsf{L}_{f_\epsilon(\overline{n})}(\tau')), s, h)$. We can apply the induction hypothesis with D_ϵ, $0 \in f(\epsilon(\overline{n}))$; Γ, $\mathsf{l}\colon \mathsf{L}_{f_\epsilon(\overline{n})}(\tau') \vdash_\Sigma e : \tau$ to obtain $Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$.

OSMatch-Cons: In this case $e = \mathsf{match\ l\ with\ |\ Nil} \Rightarrow e_1 \mid \mathsf{Cons(hd,\ tl)} \Rightarrow e_2$ for some l, hd, tl, e_1 and e_2. The typing context has the form $\Gamma = \Gamma' \cup \{\mathsf{l} : \mathsf{L}_{f(\overline{n})}(\tau')\}$ for some Γ', τ' and f. From the operational semantics we know that $h.s(\mathsf{l}).hd = v_{hd}$ and $h.s(\mathsf{l}).tl = v_{tl}$ for some v_{hd} and v_{tl}, that is $s(\mathsf{l}) \neq \mathtt{NULL}$. Due to the validity of $s(\mathsf{l})$ and Lemma 2.1, there exists $n_0 \geq 1 \in f(\epsilon(\overline{n}))$. From the validity $s(\mathsf{l}) \models^h_{\mathsf{L}_{f(\epsilon(\overline{n}))}(\tau'_{\eta\epsilon})} w_{hd} : w_{tl}$ the validities of v_{hd} and v_{tl} follow: $v_{hd} \models^h_{\tau'_{\eta\epsilon}} w_{hd}$, $v_{tl} \models^h_{\mathsf{L}_{f_\epsilon(\overline{n})-1}(\tau'_{\eta\epsilon})} w_{tl}$.

From $Valid_{\mathsf{store}}(dom(s), \Gamma_{\eta\epsilon}, s, h)$ and the results above, we obtain

$$Valid_{\mathsf{store}}(dom(s'), \Gamma_{\eta\epsilon},\ \mathsf{l}\colon \mathsf{L}_{f(\epsilon(\overline{n}))}(\tau'_{\eta\epsilon}),\ \mathsf{hd}\colon \tau'_{\eta\epsilon},\ \mathsf{tl}\colon \mathsf{L}_{f(\epsilon(\overline{n}))-1}(\tau'_{\eta\epsilon}), s', h)$$

where $s' = s[\mathsf{hd} := v_{hd}][\mathsf{tl} := v_{tl}]$. From the typing rule for e we obtain that

$$D, n_0 \geq 1 \in f(\overline{n});\ \Gamma',\ \mathsf{l}\colon \mathsf{L}_{f(\epsilon(\overline{n}))}(\tau'_{\eta\epsilon}),\ \mathsf{hd}\colon \tau'_{\eta\epsilon},\ \mathsf{tl}\colon \mathsf{L}_{f(\epsilon(\overline{n}))-1}(\tau'_{\eta\epsilon}) \vdash_\Sigma e_2 : \tau_{\eta\epsilon}$$

With $\epsilon' = \epsilon[n_0 := \mathit{length}_h(s(\mathsf{l}))]$ the induction hypothesis yields

$$\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s'), \left\{\begin{array}{l} \Gamma'_{\eta\epsilon} \cup \\ \{\mathsf{l}\colon \mathsf{L}_{f(\epsilon'(\overline{n}))}(\tau'_{\eta\epsilon'}), \\ \ \mathsf{hd}\colon \tau'_{\eta\epsilon'}, \\ \ \mathsf{tl}\colon \mathsf{L}_{f(\epsilon'(\overline{n}))-1}(\tau'_{\eta\epsilon'})\} \end{array}\right\}, s\left[\begin{array}{l}\mathsf{hd} := v_{hd}, \\ \mathsf{tl} := v_{tl}\end{array}\right], h) \implies$$
$$\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon'}, h')\,.$$

Now from the induction hypothesis and the fact that $n_0 \notin SV(\tau)$ (and thus, $\tau_{\eta\epsilon} = \tau_{\eta\epsilon'}$), we have $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$.

OSFun: We want to apply the induction assumption to

$$[y_1 := v_1, \ldots, y_k := v_k];\ h;\ \mathcal{C} \vdash e_f \rightsquigarrow v;\ h'.$$

Since the original typing judgement is a node in a derivation tree, where all called in e functions are defined via letfun, there must be a node in the derivation tree with True, $y_1 : \tau^\circ, \ldots, y_k : \tau^\circ_k \vdash_\Sigma e_f : \tau_0$. Trivially, the domains of the frame store $[y_1 := v_1, \ldots, y_k := v_k]$ and the context $y_1 : \tau^\circ, \ldots, y_k : \tau^\circ_k$ coincide.
We take η' and ϵ', such that:

- $\eta'(\alpha) = \eta(\tau_\alpha)$, where τ_α is such that α is replaced by τ_α in the instantiation σ of the signature in *this* application of the FunApp-rule.
- $\epsilon'(n_{ij}) = \epsilon(f_{ij})$, where n_{ij} is replaced by f_{ij} in the instantiation σ of the signature in *this* application of the FunApp-rule.

True ("no conditions") holds trivially on ϵ'. From the induction assumption we have

$$\mathit{Valid}_{\mathsf{store}}((y_1, \ldots y_k), (y_1 : \tau^\circ_{1\,\eta'\epsilon'}, \ldots, y_k : \tau^\circ_{k,\,\eta'\epsilon'}), [y_1 := v_1, \ldots, y_n := v_n], h)$$
$$\implies \mathit{Valid}_{\mathsf{val}}(v, \tau_{0\,\eta'\epsilon'}, h')$$

From $\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s), \Gamma_{\eta\epsilon}, s, h)$ we have the validity of the values of the actual parameters: $v_i \models^h_{\Gamma_{\eta\epsilon}(\mathsf{l}_i)} w_i$ for some w_i, where $1 \le i \le k$. Since $\Gamma_{\eta\epsilon}(y_i) = \tau^\circ_{i\,\eta'\epsilon'}$, the left-hand side of the implication holds, and one obtains $\mathit{Valid}_{\mathsf{val}}(v, \tau_{0\,\eta'\epsilon'}, h')$. It is easy to see that

$$\eta\epsilon(\sigma(\tau_0)) = \eta\epsilon(\tau_0[\ldots\alpha := \tau_\alpha\ldots][\ldots n_{ij} := f_{ij}\ldots]) =$$
$$\tau_0[\ldots\alpha := \eta(\tau_\alpha)\ldots][\ldots n_{ij} := \epsilon(f_{ij})\ldots] = \tau_{0\,\eta'\epsilon'}$$

Therefore, we obtain $\mathit{Valid}_{\mathsf{val}}(v, \eta(\epsilon(\sigma(\tau_0))), h')$ and using the rule $D \vdash \tau \to \sigma(\tau_0)$ we obtain $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$ by Lemma 2.2.

□

Bibliography

[AAG+07] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost analysis of Java bytecode. In Rocco De Nicola, editor, *Proceedings of the 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 157–172. Springer-Verlag, March 2007. Cited on pages 87 and 135.

[AAGP08] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In María Alpuente and Germán Vidal, editors, *Proceedings of the 15th international symposium on Static Analysis (SAS'08)*, volume 5079 of *LNCS*, pages 221–237, Valencia, Spain, July 16–18, 2008, 2008. Springer. Cited on pages 87 and 135.

[AAGP09] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Cost relation systems: A language-independent target language for cost analysis. *Electronic Notes in Theoretical Computer Science*, 248:31–46, 2009. Cited on pages 87 and 135.

[AAMS10] David Aspinall, Robert Atkey, Kenneth MacKenzie, and Donald Sannella. Symbolic and analytic techniques for resource analysis of Java bytecode. In *Proceedings of the 5th international symposium on Trustworthy Global Computing (TGC'10)*, LNCS, pages 1–22, Berlin, Heidelberg, 2010. Springer-Verlag. Cited on pages 84 and 139.

[Abe06] Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians University, Munich, 2006. Cited on pages 81 and 113.

[Abe09] Andreas Abel. Implementing a normalizer using sized heterogeneous types. *Journal of Functional Programming*, 19:287–310, July 2009. Cited on page 113.

[ABG+11] Elvira Albert, Richard Bubel, Samir Genaim, Reiner Hähnle, Germán Puebla, and Guillermo Román. Verified resource guarantees using COSTA and KeY. In *Proceedings of the 2011 ACM SIGPLAN workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'11)*. ACM, January 2011. To appear. Cited on page 87.

[ABHS07] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. KeY: a formal method for object-oriented systems. In *Proceedings of the 9th IFIP WG 6.1 international conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'07)*, LNCS, pages 32–43, Paphos, Cyprus, 2007. Springer-Verlag. Cited on pages 6 and 87.

[ABT07] Vincent Atassi, Patrick Baillot, and Kazushige Terui. Verification of PTIME reducibility for System F terms: Type inference in dual light affine logic. *Logical Methods in Computer Science*, 3(4), 2007. Cited on page 136.

[ACGDL04] Roberto M. Amadio, Solange Coupet-Grimal, Silvano Dal Zilio, and Jakubiec Line. A functional scenario for bytecode verification of resource bounds. In *Proceedings of the 18th international workshop on Computer Science Logic (CSL'04)*, volume 3210 of *LNCS*, pages 265–279. Springer, 2004. Cited on page 92.

[AD06] Roberto M. Amadio and Silvano Dal Zilio. Resource control for synchronous cooperative threads. *Theoretical Computer Science*, 358(2-3):229–254, August 2006. Cited on page 92.

[ADG08] Irem Aktug, Mads Dam, and Dilian Gurov. Provably correct runtime monitoring. In *Proceedings of the 15th international symposium on Formal Methods (FM'08)*, LNCS, pages 262–277, Turku, Finland, 2008. Springer-Verlag. Cited on page 54.

[AFL95] Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming Language Design and Implementation (PLDI'95)*, pages 174–185, La Jolla, California, USA, 1995. ACM. Cited on page 88.

[AGG09] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Live heap space analysis for languages with garbage collection. In *Proceedings of the 2009 International Symposium on Memory Management (ISMM'09)*, pages 129–138. ACM, 2009. Cited on page 87.

[AGH05] Ken Arnold, James Gosling, and David Holmes. *The Java Language Specification, fourth edition.* The Java Series. Prentice Hall, 2005. Cited on page 52.

[AHL+08] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. The Verisoft approach to systems verification. In *Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments (VSTTE '08)*, LNCS, pages 209–224, Berlin, Heidelberg, 2008. Springer-Verlag. Cited on page 36.

[Akt08] Iren Aktug. *Algorithmic Verification Techniques for Mobile Code.* PhD thesis, Royal Institute of Technology, Sweden, 2008. Cited on page 54.

[AM06] David Aspinall and Kenneth MacKenzie. Mobile resource guarantees and policies. In Gilles Barthe, Benjamin Grégoire, Marieke Huisman, and Jean-Louis Lanet, editors, *Revised selected papers of the 2nd international workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'05)*, volume 3956 of *LNCS*, pages 16–36. Springer, 2006. Cited on page 136.

[Ama03] Roberto M. Amadio. Max-plus quasi-interpretations. In M. Hofmann, editor, *Proceedings of the 6th international conference on Typed Lambda Calculi and Applications (TLCA'03)*, volume 2701 of *LNCS*, pages 31–45, Valencia, Spain, June 10-12 2003. Springer. Cited on page 92.

[Ama05] Roberto M. Amadio. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae*, 65(1-2):29–60, August 2005. Cited on pages 92, 113 and 135.

[AN08] Irem Aktug and Katsiaryna Naliuka. ConSpec: A formal language for policy specification. In *Run Time Enforcement for Mobile and Distributed*

Systems (REM'07), volume 197-1 of *Electronic Notes in Theoretical Computer Science*, pages 45–58. Elsevier, February 2008. Cited on page 54.

[ANN99] Torben Amtoft, Hanne R. Nielson, and Flemming Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999. Cited on page 13.

[Ann05] Saravanan Annamalai. Verification of the Fiasco IPC implementation. Master's thesis, Dresden University of Technology, December 2005. Cited on page 35.

[Atk10] Robert Atkey. Amortised resource analysis with separation logic. In Andrew D. Gordon, editor, *Proceedings of the 19th European Symposium on Programming (ESOP'10)*, volume 6012 of *LNCS*, pages 85–103. Springer, 2010. Cited on pages 84 and 139.

[BBHI05] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling: meta-level guidance for mathematical reasoning*. Cambridge University Press, 2005. Cited on page 71.

[BCO06] Josh Berdine, Cristiano Calcagno, and Peter O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Proceedings of the 4th international symposium on Formal Methods for Components and Objects (FMCO'05)*, volume 4111 of *LNCS*, pages 115–137. Springer, 2006. Cited on page 73.

[Ben01] Ralph Benzinger. Automated complexity analysis of Nuprl extracted programs. *Journal of Functional Programming*, 11:3–31, January 2001. Cited on page 86.

[Ben04] Ralph Benzinger. Automated higher-order complexity analysis. *Theoretical Computer Science*, 318(1-2):79–103, 2004. Cited on page 86.

[BFHH07] A. Bonenfant, C. Ferdinand, K. Hammond, and R. Heckmann. Worst-case execution times for a purely functional language. *Implementation and Application of Functional Languages*, pages 235–252, 2007. Cited on page 87.

[BGR08] Gilles Barthe, Benjamin Grégoire, and Colin Riba. Type-based termination with sized products. In *Proceedings of the 22nd international workshop on Computer Science Logic (CSL'08)*, LNCS, pages 493–507, Bertinoro, Italy, 2008. Springer-Verlag. Cited on page 81.

[BH05] Edwin Brady and Kevin Hammond. A dependently typed framework for static analysis of program execution costs. In *Revised selected papers of the 17th international symposium on Implementation and Application of Functional Languages (IFL'05)*, pages 74–90. Springer, 2005. Cited on page 90.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2007. Cited on page 138.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, May 2008. Cited on page 5.

[BMM04] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. On complexity analysis by quasi-interpretation. In *2nd Appsem II workshop (APPSEM'04)*, pages 85–95. none, Tallinn, Estonia, 2004. Cited on pages 91, 92, 113 and 135.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998. Cited on page 91.

[Bor00] Richard Bornat. Proving pointer programs in Hoare logic. In *Proceedings of the 5th international conference on Mathematics of Program Construction (MPC'00)*, volume 1837, pages 102–126, London, UK, 2000. Springer-Verlag. Cited on pages 60 and 72.

[BP05] Javier Blanco and Castro Pablo. A semantics for proving class correctness. unpublished, 2005. Cited on page 72.

[BS93] Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems. In *Proceedings of the 13th conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'03)*, pages 41–51, London, UK, 1993. Springer-Verlag. Cited on pages 13 and 141.

[BS96] Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1996. Cited on pages 13 and 141.

[BTV96] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'96)*, pages 171–183, St. Petersburg Beach, Florida, USA, 1996. ACM. Cited on page 88.

[Bur72] Rodney Burstall. Some techniques for proving correctness of programs which alter data structures. In *Machine Intelligence*, volume 7, pages 22–50. Edinburgh University Press, 1972. Cited on pages 60 and 73.

[BW90] Michael Barr and Charles Wells. *Category theory for computing science.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990. Cited on page 98.

[BW01] Alan Burns and Andy Welling. *Real-Time Systems and Programming Languages (Third Edition).* Addison Wesley Longman, March 2001. Cited on page 14.

[Cam08] Brian Campbell. *Space Cost Analysis Using Sized Types.* PhD thesis, School of Informatics, University of Edinburgh, 2008. Cited on pages 84 and 113.

[Cam09] Brian Campbell. Amortised memory analysis using the depth of data structures. In *Proceedings of the 18th European Symposium on Programming (ESOP'09)*, pages 190–204, York, UK, 2009. Springer-Verlag. Cited on page 84.

[CBF91] S. Chatterjee, G. E. Blelloch, and A. L. Fischer. Size and access inference for data-parallel programs. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming Language Design and Implementation (PLDI'91)*, pages 130–144, Toronto, Ontario, Canada, 1991. ACM Press. Cited on page 114.

[CCF+05] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE Analyser. In M. Sagiv, editor, *Proceedings of the 14th European Symposium on Programming (ESOP'05)*, volume 3444 of *LNCS*, pages 21–30, Edinburgh, Scotland, April 2–10 2005. Springer. Cited on page 91.

[CK01] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14:261–300, September 2001. Cited on pages 79, 80, 83 and 90.

[CKS08] David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In *Proceedings of the 21st International*

Conference on Theorem Proving in Higher Order Logics (TPHOLs'08), volume 5170 of *LNCS*, pages 167–182, Montreal, Canada, August 2008. Springer-Verlag. Cited on page 35.

[CKX03] Wei-Ngan Chin, Siau-Cheng Khoo, and Dana N. Xu. Extending sized type with collection analysis. In *Proceedings of the 2003 ACM SIGPLAN workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'03)*, pages 75–84, San Diego, California, USA, 2003. ACM. Cited on page 80.

[CL87] Charles K. Chui and Ming-Jun Lai. Vandermonde determinants and Lagrange interpolation in R^s. *Nonlinear and convex analysis*, pages 23–35, 1987. Cited on page 128.

[CNPQ08] Wei-Ngan Chin, Huu Hai Nguyen, Corneliu Popeea, and Shengchao Qin. Analysing memory resource bounds for low-level programs. In *Proceedings of the 7th International Symposium on Memory Management (ISMM'08)*, pages 151–160, Tucson, AZ, USA, 2008. ACM. Cited on page 86.

[CNQM05] Wei-Ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Rinard Martin. Memory usage verification for OO programs. In C. Hankin and I. Siveroni, editors, *Proceedings of the 12th international symposium on Static Analysis (SAS'05)*, volume 3672 of *LNCS*, pages 70–86. Springer, 2005. Cited on page 113.

[Coo72] D.C. Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 7(91–99):300, 1972. Cited on page 78.

[CP07] Yoonsik Cheon and Ashaveena Perumandla. Specifying and checking method call sequences of Java programs. *Software Quality Journal*, 15:7–25, March 2007. Cited on page 54.

[CW00] Karl Crary and Stephnie Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'00)*, pages 184–198, Boston, MA, USA, 2000. ACM. Cited on page 90.

[Dan08] Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the 35th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'08)*, pages 133–144, San Francisco, California, USA, 2008. ACM. Cited on page 90.

[DCB11] Benjamin Delaware, William R. Cook, and Don Batory. Product lines of theorems. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'11)*, Portland, Oregon, USA, 2011. ACM. To appear. Cited on page 139.

[DDB08] Matthias Daum, Jan Dörrenbächer, and Sebastian Bogan. Model stack for the pervasive verification of a microkernel-based operating system. In B. Beckert and G. Klein, editors, *Proceedings of the 5th International Verification Workshop in connection with IJCAR 2008*, volume 372 of *CEUR Workshop Proceedings*, pages 56–70, Sydney, Australia, 2008. CEUR-WS.org. Cited on page 36.

[DDH72] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured programming*. Academic Press, London, UK, 1972. Cited on page 15.

[dDMP10] Javier de Dios, Manuel Montenegro, and Ricardo Peña. Certified absence of dangling pointers in a language with explicit deallocation. In

Dominique Méry and Stephan Merz, editors, *Proceedings of the 8th international conference on, Integrated Formal Methods (IFM'10)*, volume 6396 of *LNCS*, pages 305–319, Nancy, France, October 11-14, 2010, 2010. Springer. Cited on page 90.

[DF10] Dino Distefano and Ivana Filipović. Memory leaks detection in Java by bi-abductive inference. In David Rosenblum and Gabriele Taentzer, editors, *Proceedings of the 13th international conference on Fundamental Approaches to Software Engineering (FASE'10)*, volume 6013 of *LNCS*, pages 278–292, Paphos, Cyprus, 2010. Springer. Cited on page 73.

[DJG92] Vincent Dornic, Pierre Jouvelot, and David K. Gifford. Polymorphic time systems for estimating program complexity. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1:33–45, March 1992. Cited on pages 13, 77 and 82.

[dM09] Maarten de Mol. *Reasoning About Functional Programs: Sparkle, a proof assistant for Clean.* PhD thesis, Radboud University Nijmegen, 2009. Cited on page 6.

[dMD06] Leonardo de Moura and Bruno Dutertre. Yices 1.0: An efficient SMT solver. *The Satisfiability Modulo Theories Competition (SMT-COMP)*, 2006. Cited on page 10.

[dMvEP08] Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer. Proving properties of lazy functional programs with SPARKLE. In Zoltán Horváth, editor, *Revised selected lectures of the 2nd Central-European Functional Programming School (CEFP 2007)*, volume 5161 of *LNCS*, pages 41–86, Cluj-Napoca, Romania, 2008. Springer. Cited on page 6.

[DT97] G.B. Dantzig and M.N. Thapa. *Linear programming: introduction.* Springer Verlag, 1997. Cited on page 83.

[dVRM08] Edsko de Vries, Plasmeijer Rinus, and Abrahamson David M. Uniqueness typing simplified. In *Revised selected papers of the 19th international symposium on Implementation and Application of Functional Languages (IFL'07)*, page 201, Freiburg, Germany, September 27-29, 2007, 2008. Springer-Verlag. Cited on pages 13 and 141.

[EKE08] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified protection model of the seL4 microkernel. In J. Woodcock and N. Shankar, editors, *Verified Software: Theories, Tools, Experiments*, volume 5295 of *LNCS*, Toronto, Canada, October 2008. Springer. Cited on page 35.

[End05] Endrawaty. Verification of the Fiasco IPC implementation. Master's thesis, Dresden University of Technology, March 2005. Cited on page 35.

[FA08] G. Finnie and P. Amey. SPARK 95 – The SPADE Ada 95 Kernel (including RavenSPARK). Technical Report 4.8, Praxis High Integrity Systems, 2008. Cited on page 15.

[FHL+01] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proceedings of the 1st international workshop on Embedded Software (EMSOFT'01)*, volume 2211 of *LNCS*, pages 469–485. Springer, 2001. Cited on pages 87 and 91.

[Fit96] Melvin Fitting. *First-Order Logic and Automated Theorem Proving (2nd ed.).* Springer, 1996. Cited on page 6.

[FM07] Jean Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th international conference on Computer Aided Verification (CAV'07)*, volume 4590 of *LNCS*, pages 173–177, Berlin, Germany, 2007. Springer-Verlag. Cited on pages 60, 73 and 138.

[FMWA99] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2-3):163–189, 1999. Cited on page 87.

[GG06] Alain Giorgetti and Julien Groslambert. JAG: JML Annotation Generation for verifying temporal properties. In *Proceedings of the 9th international conference on Fundamental Approaches to Software Engineering (FASE'06)*, volume 3922 of *LNCS*, pages 373–376, Vienna, Austria, March 2006. Springer. Cited on page 54.

[GL86] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM conference on LISP and functional programming (LFP'86)*, pages 28–38, Cambridge, Massachusetts, USA, 1986. ACM. Cited on page 13.

[GL02] Gustavo Gómez and Yanhong A. Liu. Automatic time-bound analysis for a higher-order language. In *Proceedings of the 2002 ACM SIGPLAN workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*, pages 75–86, Portland, Oregon, USA, 2002. ACM. Cited on page 91.

[GMC09] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'09)*, pages 127–139, Savannah, GA, USA, 2009. ACM. Cited on page 91.

[GMR08] Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. A logical account of PSPACE. In *Proceedings of the 35th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'08)*, pages 121–131, San Francisco, January 10-12, 2008, 2008. Cited on page 136.

[Gro01] Bernd Grobauer. Cost recurrences for dml programs. In *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, pages 253–264, Florence, Italy, 2001. ACM. Cited on page 90.

[HAH11] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *Proceedings of the 38th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'11)*, 2011. To appear. Cited on page 85.

[Ham02] Kevin Hammond. Hume: a bounded time concurrent language. In *Proceedings of the 7th IEEE International Conference on Electronics, Circuits and Systems (ICECS'00)*, volume 1, pages 407–411. IEEE, 2002. Cited on pages 84 and 87.

[HBH+07] Christoph A. Herrmann, Armelle Bonenfant, Kevin Hammond, Steffen Jost, Hans-Wolfgang Loidl, and Robert Pointon. Automatic amortised worst-case execution time analysis. In *Proceedings of the 7th International*

Workshop on Worst-Case Execution Time Analysis (WCET'07), pages 13–18, Pisa, Italy, July 3, 2007, 2007. Cited on page 85.

[HFH06a] Kevin Hammond, Christian Ferdinand, and Reinhold Heckmann. Towards formally verifiable resource bounds for real-time embedded systems. *ACM SIGBED Review*, 3(4):27–36, 2006. Cited on page 84.

[HFH+06b] Kevin Hammond, Christian Ferdinand, Reinhold Heckmann, Roy Dyckhoff, Martin Hofmann, Steffen Jost, Hans-Wolfgang Loidl, Greg Michaelson, Robert Pointon, Norman Scaife, Jocelyn Sérot, and Andy Wallace. Towards formally verifiable WCET analysis for a functional programming language. In Frank Mueller, editor, *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, OASICS, Dresden, Germany, 2006. Internationales Begegnungs- und Forschungszentrum f"ur Informatik (IBFI). Cited on page 85.

[HH01] Michael Hohmuth and Hermann Härtig. Pragmatic nonblocking synchronization for real-time systems. In *Proceedings of the USENIX Annual Technical Conference, General Track*, pages 217–230, Berkeley, CA, USA, 2001. USENIX Association. Cited on pages 22 and 34.

[HH10a] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polymorphic recursion and partial big-step operational semantics. In Kazunori Ueda, editor, *Proceedings of the 8th Asian Symposium on Programming Languages and Systems (APLAS'10)*, volume 6461 of *LNCS*, pages 172–187. Springer, 2010. Cited on page 85.

[HH10b] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential: A static inference of polynomial bounds for functional programs. In *Proceedings of the 19th European Symposium on Programming (ESOP'10)*, volume 6012 of *LNCS*, pages 287–306. Springer, 2010. Cited on page 85.

[HHM07] Jurriaan Hage, Stefan Holdermans, and Arie Middelkoop. A generic usage analysis with subeffect qualifiers. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*, pages 235–246, Freiburg, Germany, 2007. ACM. Cited on pages 13 and 14.

[HHW98] Hermann Hartig, Michael Hohmuth, and Jean Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel and Real-Time Systems (PART'98)*, 1998. Cited on page 24.

[HJ99] Charles Antony Richard Hoare and He Jifeng. A trace model for pointers and objects. In Rachid Guerraoui, editor, *Proceedings of the 13th European Conference on European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *LNCS*, pages 1–17. Springer, 1999. Cited on page 72.

[HJ03] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. *SIGPLAN Not.*, 38(1):185–197, 2003. Cited on pages 81, 84, 112 and 141.

[HJ06] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis (for an object-oriented language). In Peter Sestoft, editor, *Proceedings of the 15th European Symposium on Programming (ESOP'06)*, volume 3924 of *LNCS*, pages 22–37, Vienna, Austria, 2006. Springer. Cited on pages 84, 105, 113 and 139.

[HL01] Christoph A. Herrmann and Christian Lengauer. A transformational approach which combines size inference and program optimization. In

Walid Taha, editor, *Proceedings of the 2nd international conference on Semantics, Applications, and Implementation of Program Generation (SAIG'01)*, volume 2196 of *LNCS*, pages 199–218, Florence, Italy, 2001. Springer-Verlag. Cited on page 113.

[HLA94] L. Huelsbergen, J.R. Larus, and A. Aiken. Using the run-time sizes of data structures to guide parallel-thread creation. *ACM SIGPLAN Lisp Pointers*, 7(3):79–90, 1994. Cited on page 91.

[HLA+05] Galen Hunt, James R. Larus, Martin Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian D. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, October 2005. Cited on page 36.

[HM03] Kevin Hammond and Greg Michaelson. Hume: a domain-specific language for real-time embedded systems. In *Proceedings of the 2nd international conference on Generative Programming and Component Engineering (GPCE'03)*, volume 2830 of *LNCS*, pages 37–56, Erfurt, Germany, 2003. Springer-Verlag. Cited on page 84.

[HM04a] Kevin Hammond and G. Michaelson. The design of Hume: a high-level language for the real-time embedded systems domain. *Domain-Specific Program Generation*, pages 5–35, 2004. Cited on page 84.

[HM04b] Nao Hirokawa and Aart Middeldorp. Polynomial interpretations with negative coefficients. In *Proceedings of the 7th international conference on Artificial Intelligence and Symbolic Computation (AISC'04)*, volume 3249 of *LNCS*, Linz, Austria, 2004. Springer. Cited on pages 92 and 136.

[HM07] Thierry Hubert and Claude Marché. Separation analysis for deductive verification. In Werner Damm and Holger Hermanns, editors, *Heap Analysis and Verification (HAV'07)*, pages 81–93, Braga, Portugal, 2007. Cited on pages 60, 72 and 73.

[HMN01] Fritz Henglein, Henning Makholm, and Henning Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 175–186, Florence, Italy, 2001. ACM. Cited on page 88.

[Hoh98] Michael Hohmuth. The Fiasco kernel: Requirements definition. Technical Report TUD-FI98-12, Dresden University of Technology, 1998. Cited on page 22.

[Hoh02] Michael Hohmuth. *Pragmatic nonblocking synchronization for real-time systems.* PhD thesis, Dresden University of Technology, September 2002. Cited on page 28.

[Hol04] Gerard J. Holzmann. *The SPIN model checker: Primer and reference manual.* Addison Wesley, 2004. Cited on page 35.

[HOP03] Engelbert Hubbers, Martijn Oostdijk, and Erik Poll. From finite state machines to provably correct Java Card applets. In D. Gritzalis, S. De Capitani di Vimercati, P. Samarati, and S.K. Katsikas, editors, *Workshop of IFIP WG 11.2 - Small Systems Security (SEC'03)*, pages 465–470. Kluwer Academic Publishers, May 2003. Cited on page 54.

[HP99] John Hughes and Lars Pareto. Recursion and dynamic data structures in bounded space: Towards embedded ML programming. In *Proceedings*

of the 4th ACM SIGPLAN International Conference on Functional Programming (ICFP'99), pages 70–81, Paris, France, 1999. ACM. Cited on pages 13, 79, 82, 83, 88 and 90.

[HP01] Michael Hohmuth and Michael Peter. Helping in a multiprocessor environment. In *Proceedings of the 2nd Workshop on Common Microkernel System Platforms*, 2001. Cited on page 22.

[HP08] Mark A. Hillebrand and Wolfgang J. Paul. On the architecture of system verification environments. In K. Yorav, editor, *Proceedings of the 3rd international Haifa Verification Conference on Hardware and Software: Verification and Testing (HVC'07)*, volume 4899 of *LNCS*, pages 153–168, Haifa, Israel, 2008. Springer. Cited on page 36.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'96)*, pages 410–423, St. Petersburg Beach, Florida, USA, 1996. ACM. Cited on pages 78, 83, 88, 90 and 105.

[HR09] Martin Hofmann and Dulma Rodriguez. Efficient type-checking for amortised heap-space analysis. In *Proceedings of the 23rd CSL international conference and 18th EACSL Annual conference on Computer science logic (CSL'09/EACSL'09)*, LNCS, pages 317–331, Coimbra, Portugal, 2009. Springer-Verlag. Cited on pages 84 and 139.

[HT03] Michael Hohmuth and Hendrik Tews. The semantics of C++ data types: Towards verifying low-level system components. In D. Basin and B. Wolff, editors, *Emerging Trends Proceedings of the 16th international conference on Theorem Proving in Higher Order Logics (TPHOLs'03)*, pages 127–144. Institut für Informatik Universität Freiburg, 2003. Technical Report No. 187. Cited on page 35.

[HT05] Michael Hohmuth and Hendrik Tews. The VFiasco approach for a verified operating system. In *Proceedings of the 2nd ECOOP Workshop on Programming Languages and Operating Systems*, Glasgow, UK, 2005. Cited on page 35.

[HT09] Marieke Huisman and Alejandro Tamalet. A formal connection between security automata and jml annotations. In *Proceedings of the 12th international conference on Fundamental Approaches to Software Engineering (FASE'09)*, volume 5503 of *LNCS*, pages 340–354, York, UK, 2009. Springer. Cited on page 3.

[Hui01] Marieke Huisman. *Reasoning about Java programs in higher order logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001. Cited on pages 6, 26 and 138.

[Hui09] Marieke Huisman. On the interplay between the semantics of Java's finally clauses and the JML run-time checker. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs (FTfJP'09)*, pages 8:1–8:6. ACM, 2009. Cited on pages 39 and 52.

[JC94] C. Barry Jay and J. Robin B. Cockett. Shapely types and shape polymorphism. In *Proceedings of the 5th European Symposium on Programming (ESOP'94)*, pages 302–316, Edinburgh, UK, 1994. Springer-Verlag. Cited on page 113.

[JG91] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium*

on Principles of programming languages (POPL'91), pages 303–310, Orlando, Florida, USA, 1991. ACM. Cited on page 78.

[JLH+09] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. "carbon credits" for resource-bounded computations using amortised analysis. In Ana Cavalcanti and Dennis R. Dams, editors, *Proceedings of the 2nd world congress on Formal Methods (FM'09)*, volume 5850 of *LNCS*, pages 354–369, Eindhoven, The Netherlands, 2009. Springer. Cited on page 84.

[JLHH10] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'10)*, pages 223–236, Madrid, Spain, January 2010. ACM. Cited on pages 83 and 85.

[JLS+09] S. Jost, H-W. Loidl, N. Scaife, K. Hammond, G. Michaelson, and M. Hofmann. Worst-case execution time analysis through types. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS'09)*, pages 13–16, Dublin, Ireland, July 2009. ACM. Work-in-Progress Session. Cited on page 85.

[Jon02] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, September 2002. Cited on page 13.

[Jos10] Steffen Jost. *Automated Amortised Analysis*. PhD thesis, Ludwig-Maximilians University, Munich, August 2010. Cited on pages 84 and 85.

[JS97] Barry C. Jay and M. Sekanina. Shape checking of array programs. In *Computing: the Australasian Theory Seminar, Australian Computer Science Communications*, volume 19, pages 113–121, 1997. Cited on page 113.

[Ken94] Andrew Kennedy. Dimension types. In *Proceedings of the 5th European Symposium on Programming: (ESOP'94)*, pages 348–362. Springer-Verlag, 1994. Cited on page 12.

[KK06] Rafal Kolanski and Gerwin Klein. Formalising the L4 microkernel API. In *Proceedings of the 12th Computing: The Australasian Theroy Symposium (CATS'06)*, pages 53–68, Darlinghurst, Australia, 2006. Australian Computer Society. Cited on page 35.

[Kle09] Gerwin Klein. Operating system verification—An overview. *Sādhanā*, 34(1):27–69, February 2009. Cited on page 35.

[KM11] Matt Kaufmann and J Strother Moore. *ACL2 Version 4.2*, January 2011. Cited on page 6.

[Knu73] D.E. Knuth. *The art of computer programming: Fundamental algorithms, volume 1*. Addison-Wesley, 1973. Cited on page 86.

[Kon03] Michal Konečny. Functional in-place update with layered datatype sharing. In *Proceedings of the 6th international conference on Typed Lambda Calculi and Applications (TLCA'03)*, pages 195–210, Valencia, Spain, 2003. Springer-Verlag. Cited on page 12.

[KPRS96] W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive closure of infinite graphs and its applications. *Languages and Compilers for Parallel Computing*, pages 126–140, 1996. Cited on page 79.

[KT] Michael Kohlhase and Carolyn Talcott. Database of existing mechanised reasoning systems, `http://www-formal.stanford.edu/clt/ARS/systems.html`. Last updated June 18, 1999. Cited on page 6.

[Lan64] P.J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308, 1964. Cited on page 89.

[LH96] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In *Proceedings of the 6th European Symposium on Programming (ESOP'96)*, LNCS, pages 219–234, London, UK, 1996. Springer-Verlag. Cited on page 139.

[Lis03] Björn Lisper. Fully automatic parametric worst-case execution time analysis. In *Proceedings of 3rd international Workshop on Worst-Case Execution Time Analysis (WCET'03)*, volume MDH-MRTC-116/2003-1-SE, pages 99–102. Department of Computer Science and Engineering, Mälardalen University, 2003. Cited on page 87.

[LJ10] Hans-Wolfgang Loidl and Steffen Jost. Improvements to a resource analysis for hume. In *Proceedings of the 1st international conference on Foundational and Practical Aspects of Resource Analysis (FOPARA'09)*, LNCS, pages 18–33, Eindhoven, the Netherlands, November 2010. Springer-Verlag. Cited on page 85.

[LPC+09] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. *JML Reference Manual (draft)*, September 2009. Cited on page 39.

[Luc87] John M. Lucassen. *Types and Effects: Towards the Integration of Functional an Imperative Programming*. PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, 1987. Cited on page 13.

[Mar03] Jean-Yves Marion. Analysing the implicit complexity of programs. *Information and Computation*, 183(1):2–18, 2003. Cited on page 92.

[Mét88] D. Le Métayer. ACE: An automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(2):248–266, 1988. Cited on page 86.

[Mey03] Bertrand Meyer. Towards practical proofs of class correctness. In Didier Bert, Jonathan P. Bowen, Steve King, and Marina Waldén, editors, *Proceedings of the 3rd international conference on Formal Specification and Development in Z and B (ZB'03)*, volume 2651 of *LNCS*, pages 359–387. Springer-Verlag, 2003. Cited on pages 4, 59, 61, 63 and 72.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978. Cited on page 12.

[MN05] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199:200–227, May 2005. Cited on pages 71 and 72.

[Mos04] Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming (JLAP'04)*, 60–61:195–228, 2004. Cited on page 139.

[MP06] Jean-Yves Marion and R. Péchoux. Resource analysis by sup-interpretation. *Functional and Logic Programming*, pages 163–176, 2006. Cited on page 92.

[MP09] Jean-Yves Marion and Romain Péchoux. Sup-interpretations, a semantic method for static analysis of program resources. *ACM Transactions on Computational Logic (TOCL)*, 10:27:1–27:31, August 2009. Cited on page 92.

[MPM05] Claude Marché and Christine Paulin-Mohring. Reasoning about Java programs with aliasing and frame conditions. In Joe Hurd and Tom Melham, editors, *Proceedings of the 18th international conference on Theorem Proving in Higher Order Logics (TPHOLs'05)*, volume 3603 of *LNCS*, pages 179–194, Oxford, UK, August 22-25, 2005, 2005. Springer. Cited on page 73.

[MPnS08] Manuel Montenegro, Ricardo Peña, and Clara Segura. A type system for safe memory management and its proof of correctness. In *Proceedings of the 10th international ACM SIGPLAN conference on Principles and Practice of Declarative Programming (PPDP'08)*, pages 152–162, Valencia, Spain, 2008. ACM. Cited on page 90.

[MPnS09] Manuel Montenegro, Ricardo Peña, and Clara Segura. A resource-aware semantics and abstract machine for a functional language with explicit deallocation. *Electronic Notes in Theoretical Computer Science*, 246:167–182, August 2009. Cited on page 90.

[MPnS10] Manuel Montenegro, Ricardo Peña, and Clara Segura. A space consumption analysis by abstract interpretation. In *Proceedings of the 1st international conference on Foundational and Practical Aspects of Resource Analysis (FOPARA'09)*, LNCS, pages 34–50, Eindhoven, The Netherlands, 2010. Springer-Verlag. Cited on page 90.

[MPPD08] Chris Male, David J. Pearce, Alex Potanin, and Constantine Dymnikov. Java bytecode verification for @NonNull types. In *Proceedings of the 17th international conference on Compiler Construction (CC'08)*, pages 229–244, Budapest, Hungary, 2008. Springer-Verlag. Cited on page 12.

[MSvE11] Ken Madlener, Sjaak Smetsers, and Marko van Eekelen. Formal component-based semantics. In M.A. Reniers and P. Sobocinski, editors, *Proceedings of the 8th Workshop on Structural Operational Semantics (SOS'11)*, volume 62 of *Electronic Proceedings in Theoretical Computer Science*, pages 17–29, 2011. Cited on page 139.

[Muñ05] César Muñoz. *The PVSio Reference Manual version 2.b (draft)*, August 2005. Cited on page 10.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'97)*, pages 106–119. ACM Press, 1997. Cited on pages 6, 38 and 90.

[NN99] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In *Correct System Design, Recent Insight and Advances*, pages 114–136, London, UK, 1999. Springer-Verlag. Cited on page 94.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis (Corrected 2nd printing)*. Springer, 1999. Cited on pages 2 and 13.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998. Cited on pages 81 and 116.

[OL82] Susan S. Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):455–495, 1982. Cited on page 2.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th international conference on Computer Aided Verification (CAV'96)*, volume

1102 of *LNCS*, pages 411–414, New Brunswick, NJ, USA, 1996. Springer Verlag. Cited on pages 6, 10, 24 and 39.

[OSRS01] Sam Owre, Natarajan Shankar, John Rushby, and Dave Stringer-Calvert. PVS language reference (version 2.4). Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, November 2001. Cited on pages 6 and 59.

[Owr06] Sam Owre. Random testing in PVS. In *Proceedings of the 1st Workshop on Automated Formal Methods (AFM'06)*, Seattle, WA, USA, 2006. Cited on page 10.

[Par98] Lars Pareto. *Sized Types.* Chalmers University of Technology, Göteborg, 1998. Dissertation for the Licentiate Degree in Computing Science. Cited on pages 79 and 113.

[Par00] Lars Pareto. *Types for Crash Prevention.* PhD thesis, Chalmers University of Technology, Göteborg, 2000. Cited on pages 89 and 90.

[PBB+04] Mariela Pavlova, Gilles Barthe, Lilian Burdy, Marieke Huisman, and Jean-Louis Lanet. Enforcing high level security properties for applets. In J.-J. Quisquater, P. Paradinas, Y. Deswarte, and A.A. El Kalam, editors, *Smart Card Research and Advanced Applications (CARDIS'04)*, pages 1–16. Kluwer, 2004. Cited on pages 54 and 55.

[Pie04] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages.* MIT Press, 2004. Cited on pages 11 and 94.

[PSM06] Ricardo Peña, Clara Segura, and Manuel Montenegro. A sharing analysis for SAFE. In Henrik Nilsson, editor, *Selected revised papers of the 7th international symposium on Trends in Functional Programming (TFP'06)*, volume 7, pages 109–128. Intellect, 2006. Cited on page 90.

[Pug91] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, page 13. ACM, 1991. Cited on pages 78 and 79.

[PvE93] Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, 1993. Cited on page 13.

[PvE98] Rinus Plasmeijer and Marko van Eekelen. Concurrent clean language report - version 1.3. Technical Report CSI–R9816, Radboud University Nijmegen, June 1998. Cited on pages 6 and 13.

[RBN10] Stan Rosenberg, Anindya Banerjee, and David A. Naumann. Local reasoning and dynamic framing for the composite pattern and its clients. In *Proceedings of the 3rd international conference on Verified Software Theories, Tools, Experiments (VSTTE'10)*, LNCS, pages 183–198, Edinburgh, UK, 2010. Springer-Verlag. Cited on pages 59 and 73.

[Reu05] René Reusner. Implementierung eines Echtzeit-IPC-Pfades mit Unterbrechungspunkten für L4/Fiasco. Master's thesis, Dresden University of Technology, July 2005. Cited on page 35.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th annual IEEE symposium on Logic in Computer Science (LICS'02)*, pages 55–74. IEEE Computer Society, 2002. Cited on pages 59, 72 and 73.

[RG94] Brian Reistad and David K. Gifford. Static dependent costs for estimating execution time. In *Proceedings of the 1994 ACM conference on LISP and functional programming (LFP'94)*, pages 65–78, Orlando, Florida, USA, 1994. ACM. Cited on pages 13, 78, 82 and 90.

[Ros89] M. Rosendahl. Automatic complexity analysis. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 144–156. ACM, 1989. Cited on page 86.

[Ros06] Kyle D. Ross. Towards an automatic complexity analysis for generic programs. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Generic Programming (WGP'06)*, pages 87–95, Portland, Oregon, USA, 2006. ACM. Cited on page 86.

[SBB+01] P. Schnoebelen, B. Bérard, M. Bidoit, F. Laroussinie, and A. Petit. Systems and software verification: Model-checking techniques and tools, 2001. Cited on page 6.

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3:30–50, February 2000. Cited on pages 38 and 54.

[Sch07] Erik G.H. Schierboom. Verification of the Fiasco IPC implementation. Master's thesis, Radboud University Nijmegen, 2007. Cited on page 35.

[SDN+04] Jonathan S. Shapiro, Michael S. Doerrie, Eric Northup, Swaroop Sridhar, and Mark Miller. Towards a verified, general-purpose operating system kernel. In *Proceedings of the NICTA Workshop on OS Verification 2004*, Sydney, Australia, October 2004. Cited on page 36.

[SKvE10] Olha Shkaravska, Rody Kersten, and Marko van Eekelen. Test-based inference of polynomial loop-bound functions. In Andreas Krall and Hanspeter Mössenböck, editors, *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ'10)*, pages 99–108, Vienna, Austria, 2010. ACM. Cited on pages 81, 87 and 139.

[Sta88] John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *IEEE Computer*, 21:10–19, October 1988. Cited on page 15.

[STT+09] Erik Schierboom, Alejandro Tamalet, Hendrik Tews, Marko van Eekelen, and Sjaak Smetsers. Preemption abstraction: A lightweight approach to modelling concurrency. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *Proceedings of the 14th international workshop on Formal Methods for Industrial Critical Systems (FMICS'09)*, volume 5825 of *LNCS*, pages 149–164, Eindhoven, The Netherlands, 2009. Springer. Cited on page 3.

[SvET08a] Olha Shkaravska, Marko van Eekelen, and Alejandro Tamalet. Collected size semantics for functional programs over lists. Technical Report ICIS–R08021, Radboud University Nijmegen, November 2008. Cited on pages 5, 118, 123 and 131.

[SvET08b] Olha Shkaravska, Marko van Eekelen, and Alejandro Tamalet. Polynomial size complexity analysis with families of piecewise polynomials. Technical Report ICIS–R08020, Radboud University Nijmegen, November 2008. Cited on pages 5, 117, 118, 119, 131, 132 and 133.

[SvET11] Olha Shkaravska, Marko van Eekelen, and Alejandro Tamalet. Collected size semantics for functional programs over lists. In Sven-Bodo Scholz, editor, *Revised selected papers of the 20th international symposium on Implementation and Application of Functional Programming (IFL'08)*, volume 5836 of *LNCS*, pages 118–137, University of Hertfordshire, UK, 2011. Springer-Verlag. Cited on pages 5, 81 and 83.

[SvKvE07a] Olha Shkaravska, Ron van Kesteren, and Marko van Eekelen. Polynomial size analysis for first-order functions. In S. Ronchi Della Rocca, editor,

Proceedings of the 8th international conference on Typed Lambda Calculi and Applications (TLCA'07), volume 4583 of *LNCS*, pages 351–365, Paris, France, 2007. Springer-Verlag. Cited on pages 4, 81, 83, 94, 95, 100, 105, 109, 117, 121, 127, 132, 133 and 135.

[SvKvE07b] Olha Shkaravska, Ron van Kesteren, and Marko van Eekelen. Polynomial size analysis of first-order functions. Technical Report ICIS–R07004, Radboud University Nijmegen, January 2007. Cited on page 100.

[SW04] Zhendong Su and Gary Wassermann. Type-based inference of size relationships for XML transformations. Technical Report CSE-2004-8, UC Davis, April 2004. Cited on page 113.

[SWM00] Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In *Proceedings of the 9th European Symposium on Programming (ESOP'00)*, pages 366–381. Springer-Verlag, 2000. Cited on page 12.

[Tam06] Alejandro Tamalet. Yet another semantics for proving class correctness. Master's thesis, Facultad de Ciencias Exactas, Ingeniería y Agrimensura, Universidad Nacional de Rosario, Argentina, April 2006. Cited on page 72.

[Tar85] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985. Cited on page 81.

[TB98] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):724–767, 1998. Cited on pages 13 and 88.

[TBEH04] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17:245–265, September 2004. Cited on page 88.

[Tea10] The Coq Development Team. *The Coq Proof Asistant: Reference Manual v8.3pl1*, December 2010. Cited on page 6.

[Tew07] Hendrik Tews. Formal methods in the Robin project: Specification and verification of the Nova microhypervisor. In Hendrik Tews, editor, *Proceedings of the C/C++ Verification Workshop*, pages 59–68, July 2007. Cited on page 35.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992. Cited on page 13.

[TJ94] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111:245–296, June 1994. Cited on page 13.

[TM11] Alejandro Tamalet and Ken Madlener. Reasoning about assignments in recursive data structures. In J. Davies, L. Silva, and A. Simão, editors, *Proceedings of the 13th Brazilian Symposium on Formal Methods (SBMF'10)*, volume 6527 of *LNCS*, pages 161–176, Natal, Brazil, 2011. Springer. Cited on page 4.

[TSvE08] Alejandro Tamalet, Olha Shkaravska, and Marko van Eekelen. A size-aware type system with algebraic data types. Technical Report ICIS–R08006, Radboud University Nijmegen, April 2008. Cited on pages 4 and 98.

[TSvE09] Alejandro Tamalet, Olha Shkaravska, and Marko van Eekelen. Size analysis of algebraic data types. In Peter Achten, Pieter Koopman, and Marco T. Morazán, editors, *Selected revised papers of the 9th international symposium on Trends in Functional Programming (TFP'08)*, pages 33–48. Intellect, 2009. Cited on pages 4, 81, 83, 133, 134 and 135.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'94)*, pages 188–201, Portland, Oregon, USA, 1994. ACM. Cited on page 88.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997. Cited on page 88.

[Tuc09] Harvey Tuch. Formal verification of c systems code. *Journal of Automated Reasoning*, 42:125–187, April 2009. Cited on page 35.

[Tue09] Thomas Tuerk. A formalisation of Smallfoot in HOL. In *Proceedings of the 22nd international conference on Theorem Proving in Higher Order Logics (TPHOLs'09)*, LNCS, pages 469–484, Munich, Germany, 2009. Springer-Verlag. Cited on page 73.

[TVW09] Hendrik Tews, Marcus Völp, and Tjark Weber. Formal memory models for the verification of low-level operating-system code. *Journal of Automated Reasoning*, 42(2-4):189–227, 2009. Cited on page 35.

[TWM95] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Proceedings of the 7th international conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 1–11, La Jolla, California, USA, 1995. ACM. Cited on page 13.

[TWV+08] Hendrik Tews, Tjark Weber, M. Völp, Erik Poll, Marko van Eekelen, and Peter van Rossum. Nova micro–hypervisor verification. Robin deliverable d13. Technical Report ICIS–R08012, Radboud University Nijmegen, May 2008. Cited on page 35.

[Vas08] Pedro B. Vasconcelos. *Space Cost Analysis Using Sized Types.* PhD thesis, School of Computer Science, University of St. Andrews, August 2008. Cited on pages 80, 113 and 136.

[vdBJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 299–312, 2001. Cited on page 6.

[vE88] Marko van Eekelen. *Parallel graph rewriting, some contributions to its theory, its implementation and its application.* PhD thesis, Radboud University Nijmegen, 1988. Cited on page 13.

[vESvK+07] Marko van Eekelen, Olha Shkaravska, Ron van Kesteren, Bart Jacobs, Erik Poll, and Sjaak Smetsers. AHA: Amortized Heap space usage Analysis. In Marco Morazán, editor, *Selected revised papers of the 8th international symposium on Trends in Functional Programming (TFP'07)*, pages 36–53, New York, USA, 2007. Intellect. Cited on pages 113 and 116.

[VH04] Pedro B. Vasconcelos and Kevin Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In P. Trinder, G. Michaelson, and R. Peña, editors, *Revised selected papers of the 15th international symposium on Implementation of Functional Languages (IFL'03)*, volume 3145 of *LNCS*, pages 86–101, Edinburgh, UK, September 8-11, 2003, 2004. Springer-Verlag. Cited on pages 80 and 113.

[vHPPR98] Friedrich W. von Henke, Stephan Pfab, Holger Pfeifer, and Harald Rueß. Case studies in meta-level theorem proving. In Jim Grundy and Malcolm C. Newey, editors, *Proceedings of the 11th international conference on Theorem Proving in Higher Order Logics (TPHOLs'98)*, volume 1479 of *LNCS*, pages 461–478. Springer-Verlag, 1998. Cited on page 72.

[vKSvE07] Ron van Kesteren, Olha Shkaravska, and Marko van Eekelen. Inferring static non-monotonically sized types through testing. In *Proceedings of 16th international workshop on Functional and (Constraint) Logic Programming (WFLP'07)*, volume 216C of *Electronic Notes in Theoretical Computer Science*, pages 45–63, Paris, France, 2007. Cited on pages 81, 95, 112, 117 and 135.

[vO01] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, University of Technology, Munich, 2001. Cited on page 44.

[Wan02] Keith Wansbrough. *Simple Polymorphic Usage Analysis*. PhD thesis, University of Cambridge, 2002. Cited on page 13.

[Weg75] Ben Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, 1975. Cited on page 86.

[Wen99] Makarius Wenzel. Isar: a generic interpretative approach to readable formal proof documents. *Theorem Proving in Higher Order Logics*, pages 840–840, 1999. Cited on page 5.

[Wen10] Makarius Wenzel. *The Isabelle/Isar Reference Manual*, June 2010. Cited on page 6.

[Wol02] Pierre Wolper. *Constructing automata from temporal logic formulas: a tutorial*, pages 261–277. Springer-Verlag, 2002. Cited on page 39.

[WOLB92] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications (2nd ed.)*. McGraw-Hill, 1992. Cited on page 6.

[Xi07] Hongwei Xi. Dependent ml an approach to practical programming with dependent types. *J. Funct. Program.*, 17:215–286, March 2007. Cited on page 90.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. *SIGPLAN Not.*, 33:249–257, May 1998. Cited on page 12.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'99)*, pages 214–227, San Antonio, Texas, United States, 1999. ACM. Cited on page 90.